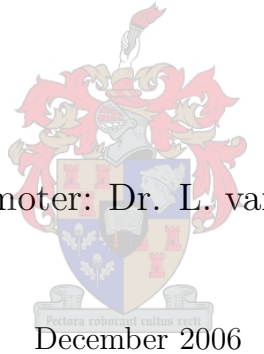
Random generation of finite automata over the domain of the regular languages.

Lesley Anne Raitt

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering at the University of Stellenbosch.



Promoter: Dr. L. van Zijl

December 2006

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously, in its entirety or in part, submitted it at any university for a degree.

Signature: _____ Date: _____

Abstract

The random generation of finite automata over the domain of their graph structures is a well-known problem. However, random generation of finite automata over the domain of the regular languages has not been studied in such detail. Random generation algorithms designed for this domain would be useful for the investigation of the properties of the regular languages associated with the finite automata.

We studied the existing enumerations and algorithms to randomly generate UDFAs and binary DFAs as they pertained to the domain of the regular languages. We evaluated the algorithms experimentally across the domain of the regular languages for small values of n and found the distributions non-uniform. Therefore, for UDFAs, we derived an algorithm for the random generation of UDFAs over the domain of the regular languages from Domaratzki *et. al.*'s [9] enumeration of the domain of the regular languages. Furthermore, for binary DFAs, we concluded that for large values of n, the bijection method is a viable means of randomly generating binary DFAs over the domain of the regular langagues.

We looked at all the random generation of union-UNFAs and $\oplus$-UNFAs across the domain of the regular languages. Our study of these UNFAs took all possible variables for the generation of UNFAs into account. The random generation of UNFAs over the domain of the regular languages is an open problem.

Opsomming

Die ewekansige generasie van eindige toestand outomate (eto's) oor die domein van hul grafiekstrukture is 'n bekende probleem. Nieteenstaande het die ewekansige generasie van eindige toestand outomate oor die domein van die regulêre tale nie soveel aandag gekry nie. Algoritmes wat eindige toestand outomate ewekansig genereer oor die domein van die regulêre tale sal nuttig wees om die ondersoek van die eienskappe van regulêre tale, wat met eto's verbind is, te bewerkstellig.

Ons het die bestaande aftellings en algoritmes bestudeer vir die ewekansige generasie van deterministiese eindige toestand outomate (deto's) met een en twee alfabetiese simbole soos dit betrekking het op die domein van die regulêre tale bestudeer. Ons het die algoritmes eksperimenteel beoordeel oor die domein van die regulêre tale vir outomate met min toestande en bevind dat die verspreiding nie eenvomig is nie. Daarom het ons 'n algoritme afgelei vir die ewekansige generasie van deto's met een alfabetsimbool oor die domein van die regulêre tale van Domaratzki *et. al.* [9] se aftelling. Bowendien, in die geval van deto's met twee alfabetsimbole met 'n groot hoeveelheid toestande is die 'bijeksie metode 'n goeie algoritme om te gebruik vir die ewekansige generasie van hierdie deto's oor die domein van die regulêre tale.

Ons het ook die ewekansige generasie van $\cup$-nie-deterministiese eindige toestand outomate en $\oplus$-nie-deterministiese eindige toestand outomate oor die domein van die regulêre tale bestudeer. Ons studie van hierdie neto's het alle moontlike veranderlikes in ageneem. Die ewekansige generering van deto's oor die domein van die regulêre tale is 'n ope probleem.

Acknowledgements

My thanks to:

- my supervisor, Dr L. van Zijl, the guiding light in this research work,
- Deon Borman, for technical support,
- my sister Gwen and my parents for their patience and support,
- my husband, Hercule du Preez and
- the National Research Foundation for financial assistance.

Contents

Chapter 1

Introduction

It is often of interest to randomly generate elements of a specific set in order to investigate common properties of that set. The elements in the set are referred to as the *domain* of the random generation. Any behavioural analysis conducted on randomly generated elements pertains only to the domain of the randomly generated elements. In order to guarantee a measure of randomness, the elements in the domain must be countable. A set is countable or *enumerable* if either it is finite or it has the same size as the set of natural numbers [25].

The current random generation algorithms for finite automata generate finite automata over different domains. These domains include the set of all connected machines and the set of all possible transition tables [18, 30]. The standard methods for the random generation of finite automata are based on the use of streams of pseudo-random integers [13, 14]. These integers are then manipulated to build a finite automaton and the finite automata are then assumed to have been randomly generated [7, 30]. However, it is important to note the domain on which the algorithm is based, as such randomly generated finite automata do not necessarily represent a (pseudo)-random selection of finite automata over the domain of the regular languages. In this thesis, our interest lies in investigating the *uniform* random generation of finite automata over the domain of the regular languages. (Throughout the rest of this thesis, we will refer to random generation over the domain of the regular languages, implying uniform random generation.)

There are many instances where it becomes interesting to investigate the properties of randomly generated finite automata as they apply to regular languages rather than to the structures of the finite automata themselves. For example, suppose one would like to experimentally investigate how many languages associated with n-state nondeterministic finite automata (NFAs) are also typically associated with deterministic finite automata (DFAs) of size $O(2^n)$. To have an indication of the theoretical results that would be expected from this problem, one could then randomly generate a large number of n-state NFAs. These NFAs would be converted to minimal DFAs. Then the percentage of languages associated with $O(2^n)$-state DFAs could be calculated. However, here one cannot assume that the NFAs, although randomly generated, are necessarily a representative sample over the regular languages.

To address this problem, we intend to study methods of randomly generating deterministic and nondeterministic finite automata, and evaluate the randomness of these sequences of finite automata over the domain of the regular languages.

1.1 Thesis outline

Chapter 2 supplies the necessary definitions of finite automata, relevant mathematical formulae and some important points concerning random number generation. As the enumeration of a domain is necessary in random generation, Chapter 3 looks at the enumeration of finite automata. Chapter 3 also examines existing methods of random generation of finite automata with particular attention to the domain. In Chapter 4, we use an existing enumeration of the number of languages accepted by n-state DFAs with a single alphabet symbol, to develop a random generation algorithm for unary DFAs across the domain of the regular languages. We analyse some of the methods of random generation of finite automata over the domain of the regular languages with the aid of some experimental results in Chapter 5. In Chapter 6, we look at the algorithms for random generation of nondeterministic finite automata over the domain of the languages accepted by n-state $\oplus$-NFAs [29]. Finally, we suggest future work in Chapter 7.

Chapter 2

Background and notation

In this chapter, we introduce the notation required for this thesis. The necessary mathematical background is given in Section 2.1, while Section 2.2 and Section 2.3 introduce the necessary background to finite automata. Section 2.4 discusses random number generation.

2.1 Mathematical background

Finite automata can be pictorially represented by graphs. Where graphs have nodes and edges, automata have states and transitions. Both are combinatorial structures. Enumeration of combinatorial structures requires some basic combinatorics [27].

2.1.1 Combinations and the binomial theorem

Some simple concepts in systematic counting are listed below. Systematic counting provides methods to obtain the total number of a set of elements without listing each element to be counted.

Factorial The definition of the *factorial*, as required for the definition of combinations, is given below:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$$

Combinations The number of subsets of size r that can be chosen from a set of n elements is $C_r^n = \frac{n!}{r!(n-r)!}$.

Example 1 *Assume we have a set of numbers $\mathcal{S} = \{0, 1, 2, 3\}$ with a size of four. Then there are C_3^4 subsets of $\mathcal{S}$ of size three:*

$$C_3^4 = \frac{4!}{3!(1)!} = \frac{4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1) \times 1} = 4.$$

The four subsets of $\mathcal{S}$ *of size three are* $\{0,1,2\},\{0,1,3\},\{0,2,3\}$ *and* $\{1,2,3\}$.

□

The total number of different subsets is also of interest to us. This can be calculated using the binomial theorem, given below.

Theorem 1 $(1+x)^n = C_0^n + C_1^n x + \ldots + C_r^n x^r + \ldots + C_n^n x^n$.

We want to know the total number of different subsets of a set with size n. This would be the sum of the sets of size zero to size n or

$$\sum_{i=1}^{n}(C_i^n).$$

We can use the binomial theorem, with $x = 1$ to calculate this value:

$$(1+1)^n = C_0^n + C_1^n + \ldots + C_r^n + \ldots + C_n^n.$$

Example 2 *Assume we have a set of numbers,* $\mathcal{S} = \{0,1\}$. *Then we know, from Theorem 1, that there are* $2^2 = 4$ *possible subsets. These subsets are* $\emptyset, \{0\}, \{1\}$ *and* $\{0,1\}$.

□

2.1.2 The Möbius function

The Möbius function is used in Domaratzki's enumeration [9] of unary DFA languages. Chapter 4 studies this enumeration in detail. The Möbius function [8, 9], represented by μ, is defined as follows:

$$\mu(n) = \begin{cases} 0, & \text{if } n \text{ is divisible by a square} > 1 \\ (-1)^s & \text{if } n = p_1 p_2 \ldots p_s, \text{ where the } p_i \text{ are distinct primes,} \end{cases} \tag{2.1}$$

for natural number n.

Example 3 *Let* $n = 25$. *Then surely* n *is divisible by a square, since* $25 = 5 \times 5$. *Hence* $\mu(25) = 0$. *Now consider* $n = 46$. *The prime factorisation of* n *is* 2×23. *These are distinct primes so* $\mu(46) = (-1)^2 = 1$. *Finally we take* $n = 273$. *The prime factorisation of* n *is* $3 \times 7 \times 13$. *Therefore* $\mu(273) = (-1)^3 = -1$.

□

We can now introduce the definitions of finite automata used in this thesis. As the various enumerations of automata use definitions which have slight differences, it is important to establish our definitions.

2.2 Deterministic finite automata

We base our definitions in this section on Sipser [25], who provides a good background to automata theory.

Definition 1 A *deterministic finite automaton (DFA)* is denoted by a five-tuple $A = (Q, \Sigma, \delta, q_0, F)$ with

- Q the finite nonempty set of states,
- Σ the finite nonempty input alphabet,
- $\delta : Q \times \Sigma \to Q$ the transition function,
- q_0 the start state, and

- $F \subseteq Q$ the set of final states.

□

As previously mentioned, the basic structure of a DFA can be seen as a graph with nodes representing states and labelled edges representing transitions. We call this the *underlying graph* of the DFA. This makes it possible to apply graph theory to investigate the properties of DFAs.

Several definitions which are important to this thesis are listed below.

Regular language A language which is recognised by a finite automaton is called a *regular language*. The language which is accepted by finite automaton M is the set of all strings that M accepts. A string u is accepted if the resultant state $\delta(q_0, u)$ is a final state.

Regular Expression Regular languages can be described by finite automata or by *regular expressions*. We assume an understanding of regular expressions. For a formal definition, see [25].

Minimal DFAs Let A be a DFA which accepts regular language L. If there is no DFA A' with fewer states which accepts L, then A is a *minimal* DFA [9].

Connected A DFA is *connected* if and only if its underlying, undirected graph is connected [12].

Strongly connected A DFA is *strongly connected* [22] if state q_i is reachable from state q_j for every $q_i, q_j \in Q$.

Accessible A DFA is *accessible* or *initially connected* [7] if and only if there exists a path from the initial state to every other state q, for $q \in Q - \{q_0\}$.

2.2.1 Unary deterministic finite automata

A unary DFA (UDFA) is a DFA with one alphabet symbol, and we assume without loss of generality that the alphabet of a UDFA is $\Sigma = \{a\}$. Throughout the rest of this thesis, we assume that all UDFAs are complete and connected. The following two theorems are based on the structure of the UDFA.

Theorem 2 *The states of any complete and connected n-state UDFA may be renumbered [20] as $q_0, q_1, \ldots, q_{n-2}, q_{n-1}$, such that state q_0 is the start state and*

$$\begin{array}{lcll} \delta(q_i, a) & = & q_{i+1}, & \text{where } 0 \leq i < n-1, \text{ and} \\ \delta(q_{n-1}, a) & = & q_k, & \text{where } 0 \leq k \leq n-1\ . \end{array}$$

□

The set of states $\{q_0, q_1, \ldots, q_{k-1}\}$ forms the tail of the UDFA. The remaining set of states $\{q_k, q_{k+1}, \ldots, q_{n-1}\}$ forms the *loop* of the UDFA. We will refer to the value k, where $\delta(q_{n-1}, a) = q_k$, as the *loop value* throughout this thesis. We assume that all UDFAs in the scope of this thesis have been renumbered such that any UDFA may be determined by its set of final states and its loop value.

To be able to easily identify minimal UDFAs, a few more UDFA specific definitions are required.

Equivalent loops Let $M_1 = \{Q_1, \Sigma_1, \delta_1, q_a, F_1\}$ and $M_2 = \{Q_2, \Sigma_2, \delta_2, q_b, F_2\}$ be two UDFAs with loop values k_1 and k_2 respectively. Then the loops of M_1 and M_2 are equivalent when machines M_1' and M_2', constructed by removing the tail states of the UDFAs, are equivalent. The tail states of M_1 are removed by constructing $M_1' = (Q', \Sigma_1, \delta_1', q_k, F')$, where $Q' = \{q_k, \ldots, q_{n-1}\}, F_1' = F_1 \backslash \{q_0, \ldots, q_{k-1}\}$ and $\delta_1'(q, a) = \delta_1(q, a)$. M_2' is constructed from M_2 in a similar manner.

Finality The *finality* of two states of an automaton is the *same* if either both are final, or both are non-final [20].

Minimal loops A loop is *minimal* if and only if that loop cannot be replaced with a shorter equivalent loop.

Minimal UDFAs are easily identified by Nicaud's characterisation theorem, given below.

Theorem 3 *An n-state UDFA A, with loop value k, is minimal [20] if and only if the following three conditions hold:*

1. *A is connected,*
2. *A has a minimal loop, and*
3. *states q_{k-1} and q_{n-1} do not have the same finality.*

□

Nondeterminism is a specialisation of determinism. The key differences between a deterministic finite automaton (DFA) and a nondeterministic finite automaton (NFA) are

- DFAs have a single start state, while NFAs have a set $Q_0 \subseteq Q$ of start states;
- the DFA transition function operates on an alphabet symbol and state and produces a single state whereas the NFA transition function operates on an alphabet symbol and state and produces a set of states; and
- DFA transitions require an alphabet symbol, while NFA transitions can include a transition on the empty symbol. For this reason the alphabet for an NFA Σ_ϵ is defined as $\Sigma \cup \{\epsilon\}$ (see Definition 2 below).

A NFA transition function takes any state q and an alphabet symbol or ϵ and produces a subset of possible next states. According to the binomial theorem (Theorem 1, page 4) there are 2^Q possible different subsets for any given set Q. In the next section, we give the formal definition of a nondeterministic finite automaton.

2.3 Nondeterministic finite automata

Definition 2 A *nondeterministic finite automaton (NFA)* is denoted by a five-tuple $N = (Q, \Sigma, \delta, Q_0, F)$ with

- Q the finite nonempty set of states,
- Σ the finite nonempty input alphabet,
- $\delta : Q \times \Sigma_\epsilon \to 2^Q$ the transition function,
- $Q_0 \subseteq Q$ the set of start states and
- $F \subseteq Q$ the set of final states. □

We rephrase the definition of *accessibility* to include NFAs:

Equivalence Let L be the language recognised by a DFA or NFA M. Then any other DFA or NFA which also recognises L, is *equivalent* to M [25].

Accessible If there exists a path within an NFA N from an initial state to state q, for every state $q \in Q - Q_0$, then N is *accessible* or *initially connected* [7].

Isomorphism Two finite automata are *isomorphic* if there is a renumbering of states such that the finite automata are identical in every way including initial and final states. More formally, finite automata $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$ are *isomorphic* if there is a one-to-one mapping α from the state set Q_1 onto Q_2 such that

$$\alpha(Q_1) \subseteq Q_2$$

$$\alpha(\delta_1(q_i, a)) = \delta_2(\alpha(q_i), a)$$

$$\alpha(F_1) \subseteq F_2$$

for all $q_i \in Q_1$ and $a \in \Sigma$. .

Pairwise nonisomorphic A set of finite automata are *pairwise nonisomorphic* if there are no two finite automata in the set which are isomorphic to each other.

Pairwise nonequivalent A set of finite automata are *pairwise nonequivalent* if there are no two finite automata in the set which are equivalent to each other.

Minimal NFAs An NFA which accepts regular language R is minimal if there is no NFA with fewer states which accepts R. If two minimal NFAs both accept regular language R, it does not imply that the two NFAs are isomorphic.

The class of *traditional* NFAs (Definition 2) is a subset of the class of the more general $\star$-*NFAs*. The $\star$-NFA is defined below.

Definition 3 A *$\star$-nondeterministic finite automaton ($\star$-NFA)* [29], is denoted by a six-tuple $N = (Q, \Sigma, \delta, q_0, F, \star)$ with

- Q the finite nonempty set of *states*,
- Σ the finite nonempty input *alphabet*,
- $\delta : Q \times \Sigma \to 2^Q$, the transition function,
- $Q_0 \subseteq Q$ the set of start states,
- $F \subseteq Q$ the set of final states and
- $\star$ any associative commutative binary operation on sets.

□

The acceptance conditions of $\star$-NFAs as used in this thesis are the same as that of the union NFA (Definition 2). The $\star$-NFA accepts a word $w \in \Sigma^*$ if for a start state $q_0 \in Q_0$, $\delta(q_0, w)$ contains at least one state which is final and therefore in set F. The traditional NFA (Definition 2) is a $\star$-NFA (Definition 3) with $\star$ taken as the union operation. The union operation is an associative, commutative binary operation. Another possible associative, commutative operation is symmetric difference ($\oplus$). In this context, $A \oplus B$ is defined as

$$(A \cup B) \backslash (A \cap B).$$

$\oplus$-NFAs will be considered in Chapter 6. For a $\oplus$-NFA,

$$\delta'(P, a) = \overset{\oplus}{_{q \in P}}\ \delta(q, a)$$

for any $a \in \Sigma$ and $P \in 2^Q$.

For every $\star$-NFA, there exists an equivalent DFA. This means that there is an equivalent DFA for every $\oplus$-NFA. Consider an NFA $N = (Q, \Sigma, \delta, Q_0, F)$. One can find an equivalent DFA

$A = (2^Q, \Sigma, \delta', q_0', F')$, by applying the *subset construction* [18]. In the subset construction, the subsets of Q are used to label the states of A. The transition function, δ' is derived from δ using the rule

$$\delta'(P, a) = \overset{\star}{_{q \in P}}\ \delta(q, a),$$

for any $a \in \Sigma$ and $P \in 2^Q$. Final states are labelled by subsets containing at least one final state of N. The start state q_0' is the state labelled by the set Q_0.

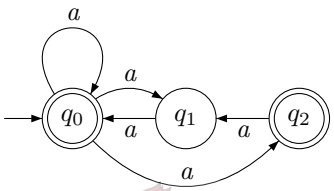
Example 4 gives a $\star$-NFA and the equivalent DFAs for $\star$ taken as union, symmetric difference and intersection.

Example 4 *Let N be a $\star$-NFA defined by*

$$N = (\{q_0, q_1, q_2\}, \{a\}, \delta, \{q_0\}, \{q_0, q_2\}, \star)$$

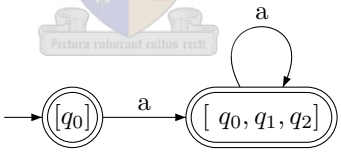
with δ given by

δ	a
q_0	$\{q_0, q_1, q_2\}$
q_1	$\{q_0\}$
q_2	$\{q_1\}$



If the $\star$ operator is taken as union in N above, an equivalent DFA can be obtained by using the subset construction as follows: the start state q_0' is $[q_0]$, as there is only one start state in N. The transition from state $[q_0]$ on alphabet symbol a is $\{q_0, q_1, q_2\} = [q_0, q_1, q_2]$. The transition from the state labelled $[q_0, q_1, q_2]$ is $\{q_0, q_1, q_2\} \cup \{q_0\} \cup \{q_1\} = [q_0, q_1, q_2]$. Therefore, the equivalent DFA obtained by the subset construction is $A_1 = \{Q_1, \{a\}, \delta_1, [q_0], \{[q_0], [q_0, q_1, q_2]\}\}$ with δ_1 given by

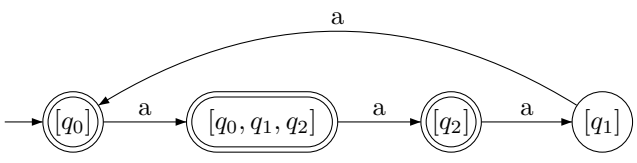
δ_1	a
$[q_0]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$



The language accepted by A_1 is a^*.

However, if the $\star$ operator in N above is taken as symmetric difference, the equivalent DFA is different. The start state q_0' is $[q_0]$, as there is only one start state in N. The transition from state $[q_0]$ on alphabet symbol a is $\{q_0, q_1, q_2\} = [q_0, q_1, q_2]$. The transition from the state labelled $[q_0, q_1, q_2]$ is $\{q_0, q_1, q_2\} \oplus \{q_0\} \oplus \{q_1\} = [q_2]$. The transition from the state labelled $[q_2]$ is $[q_1]$ and the transition from $[q_1]$ is $[q_0]$. Therefore, the DFA A_2 obtained by subset construction is $A_2 = \{Q_2, \{a\}, \delta_2, [q_0], \{[q_0], [q_2], [q_0, q_1, q_2]\}\}$ with δ_2 given by

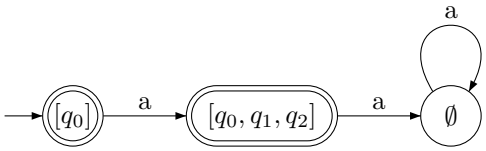
δ_2	a
$[q_0]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_2]$
$[q_2]$	$[q_1]$
$[q_1]$	$[q_0]$



The language accepted by A_2 is $(a^4)^* + a(a^4)^* + aa(a^4)^*$.

Lastly, if the $\star$ operator is taken as intersection, an equivalent DFA can be obtained by the subset construction as follows: the start state q_0' is $[q_0]$, as there is only one start state in N. The transition from state $[q_0]$ on alphabet symbol a is $\{q_0, q_1, q_2\} = [q_0, q_1, q_2]$. The transition from the state labelled $[q_0, q_1, q_2]$ is $\{q_0, q_1, q_2\} \cap \{q_0\} \cap \{q_1\} = \emptyset$. Therefore, the DFA A_2 obtained by subset construction is $A_3 = \{Q_3, \{a\}, \delta_3, [q_0], \{[q_0], [q_0, q_1, q_2]\}\}$ with δ_3 given by

δ_3	a
$[q_0]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$\emptyset$
$\emptyset$	$\emptyset$



The language accepted by A_3 is $\epsilon + a$.

□

Unless explicitly stated, we work with NFAs with a single start state and no transitions on the empty symbol. We do this so that we are able to use Domaratzki *et al.*'s results in [9].

2.4 Random numbers

Random numbers are used in many contexts and across many disciplines. Applications range from mathematical simulation to lottery draws. The demand for random numbers has led to extensive research in this field [13, 14, 16, 17].

Random numbers may be produced by physical devices or algorithms. Physical devices are typically cumbersome to use, not generally available and may produce unsatisfactory outputs [14, 17]. Algorithmically produced outputs are easily available and simple to use. However, any algorithmically produced sequence of numbers is eventually periodic [14]. This implies that the numbers will repeat in a previously occurring order. The length of the period refers to the number of elements generated before the sequence repeats. Some algorithms produce sequences with periods that are too short to be random and thus not all algorithms produce viable random numbers. It is essential to establish the desired properties of any algorithm and its resulting random number stream to be sure that it is suitable for use.

2.4.1 Desirable properties of random number streams

As random numbers are frequently used to generate random structures, it is important to define the characteristics of a good random number generator.

- **"True" randomness:** algorithmically produced sequences are not random in the true sense of the word. These sequences are therefore called *pseudo-random*. We will use the term *random* as a simplification [17] of pseudo-random. The most important requirement of such a sequence is that it should appear to be random for all practical purposes.

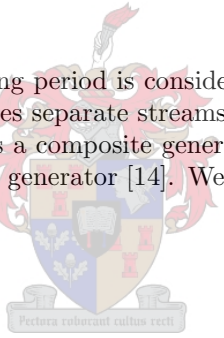
  Ideally, no statistical test or computer program should be able to tell the difference between an algorithmically produced random number output sequence $(u_0, u_1, \ldots)$ and an infinite

sequence of independent and identically distributed (i.i.d.) random variables with a probability larger than $\frac{1}{2}$. In practice, this is not actually possible as for any periodic sequence, if enough computer time is allowed it is always possible to find a statistical test for which it will fail [15].

- **Long period:** as the output from algorithmically generated number sequences are periodic [14], it is important to be sure that the period is long enough for the required purpose. The length acceptable is specific to the application of the numbers. If many numbers are required, generators with period lengths over 2^{200} are now available [17].
- **Efficiency:** ideally the implementation of the algorithm to produce the random number sequence should be fast, with low storage requirements.
- **Repeatability:** it is useful to many applications to be able to reproduce a random number sequence exactly. Results may be verified and the same stream of numbers may be used in different contexts. Algorithms to generate random numbers often require an initial value or *seed*. To generate two identical random number sequences, the same seed must be used. To change the random number sequence, the seed must be altered.

A good random number generator is necessary if the random number sequence is to be manipulated to randomly build finite automata. We used two generators for the work in this thesis. These were ranlib.c [5], to generate real numbers between zero and one, and the Mersenne Twister [19] to generate integers.

A random number generator with a long period is considered to be an adequate substitute for a random number generator that produces separate streams of random numbers. The use of separate streams of random numbers forms a composite generator. Composite generators sometimes have a shorter period then the original generator [14]. We use separate streams as recommended by [29].



Chapter 3

Related work

In this chapter, we discuss existing enumerations of finite automata. In particular, we consider the regular languages associated with n-state DFAs and NFAs in these enumerations. In the latter part of the chapter, we discuss existing methods for the random generation of finite automata.

3.1 Enumeration of automata

The enumeration of finite automata is a well-known problem: it occurred as problem 19 in Harary's 1960 list of unsolved problems [11]. Several solutions to this problem have been put forward in the literature [12, 11, 22, 23] and we consider some of these solutions in this section.

The enumeration or counting of finite automata differs from the enumeration of the regular languages. To randomly generate finite automata over the domain of the regular languages, we require an enumeration of the regular languages associated with n-state DFAs or NFAs (see page 1). Currently, such an enumeration exists only for unary DFAs [9]. Hence, we consider different known enumerations of finite automata and compare them to the domain of the regular languages accepted by n-state DFAs and NFAs. Note that the differences in the enumerations occur due to different restrictions on the finite automata in question.

The first DFA enumeration we consider is that of Harrison [12]. This enumeration ignores both initial and final states in the finite automata. Harary and Palmer [11] improved Harrison's method by incorporating initial and final states in the finite automata. However, both of these enumerations include machines which are not connected. Radke [22] enumerated strongly connected DFAs but since there are a number of accessible machines which are not strongly connected, his enumeration is, in fact, too restrictive for our purposes. Finally, Robinson [23] enumerated accessible DFAs, which is closest to the enumeration we require in this work. We will discuss each of these methods in more detail and specifically compare the enumerated domain to the domain of the regular languages.

For the comparison of the domains, an exact enumeration of the domain of the regular languages would be ideal. As noted previously, an exact enumeration (given by Domaratzki *et al.* [9]) of the number of distinct regular languages accepted by n-state finite automata only exists for UDFAs. We use this enumeration in Chapter 4 to form the basis of random generation of UDFAs across the domain of the regular languages. Domaratzki *et al.* also set bounds for the number of distinct regular languages accepted by n-state DFAs with k alphabet symbols and the number of distinct regular languages accepted by n-state NFAs. Note that the bounds are derived, but no explicit enumeration is given.

We will now consider a more detailed review of the DFA enumerations mentioned previously.

Harrison [12] derived one of the first (partial) solutions to the enumeration of automata. In the context of Harrison's work, let $\Sigma_k = \{\sigma_0, \ldots, \sigma_{k-1}\}$ be the input alphabet and $\Pi_p = \{\pi_0, \ldots, \pi_{p-1}\}$ the output alphabet. Then Harrison defined a *restricted* finite automaton as a three tuple $S = \langle S, f, g \rangle$, with

- $S = \{s_0, \ldots, s_{n-1}\}$, a set of states,
- f, a transition function which maps $S \times \Sigma_k \to S$, and
- g, the output function which maps $S \times \Sigma_k \to \Pi_p$.

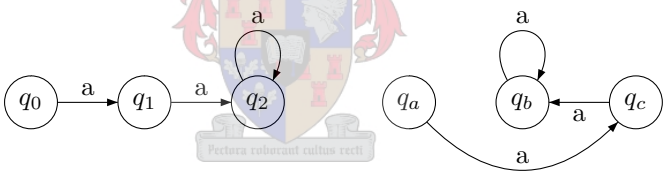
Note that Harrison's restricted definition does not include initial and final states. Therefore, this enumeration is not directly applicable to regular languages. Harrison restricted his enumerations to complete DFAs.

Harrison defines three equivalence classes of automata and then proceeds to enumerate them. The three equivalence classes are listed below.

1. Two finite automata are *isomorphic* if there is a renumbering of states such that the automata are identical. More formally, machines $A_1 = \langle S_1, f_1, g_1 \rangle$ and $A_2 = \langle S_2, f_2, g_2 \rangle$ are isomorphic if a one-to-one mapping α exists from the state set S_1 onto S_2 such that

$$\begin{aligned} \alpha(f_1(s, \sigma)) &= f_2(\alpha(s), \sigma), \quad \text{and} \\ g_1(s, \sigma) &= g_2(\alpha(s), \sigma). \end{aligned}$$

   The two finite automata below are isomorphic. This can clearly be seen, as α maps q_0 to q_a, q_1 to q_c and q_2 to q_b.



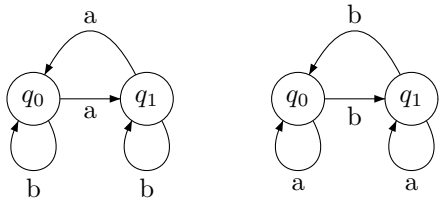
   A_1 $\qquad\qquad$ A_2

   On comparing isomorphic finite automata to the domain of the regular languages, we see that finite automata which are isomorphic accept the same language. Therefore, from the perspective of the domain of the regular languages, we want to enumerate machines such that no two are isomorphic.

2. Two automata are *equivalent with respect to an input permutation* if there is a relabelling of transitions and states such that the automata are identical. More formally, finite automata $A_1 = \langle S_1, f_1, g_1 \rangle$ and $A_2 = \langle S_2, f_2, g_2 \rangle$ are equivalent with respect to an input permutation, if α, an element in the symmetric group of degree n and β, an element in the symmetric group of degree k, exist such that

$$\begin{aligned} \alpha f_1(s, \sigma) &= f_2(\alpha(s), \beta(\sigma)), \quad \text{and} \\ g_1(s, \sigma) &= g_2(\alpha(s), \beta(\sigma)). \end{aligned}$$

The two binary automata below are equivalent with respect to an input permutation. It is easy to see that replacing all a's with b's and all b's with a's in A_2 causes A_2 to be identical to A_1.



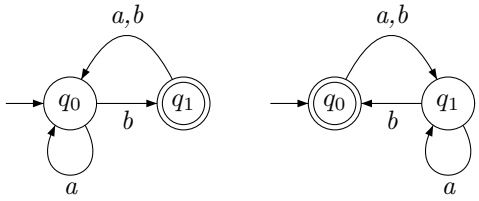
A_1 A_2

When considering the domain of the regular languages, we do not consider machines that are equivalent with respect to input permutations as equivalent. This is because machines which are equivalent with respect to input permutations can recognise different languages (such as A_1 and A_2 above).

3. Two automata are said to be equivalent with respect to both input and output permutations if there is a relabelling of the input function, output function and states such that the automata are identical. We consider only automata without output in this thesis. Note that a DFA defined without output may be seen as a DFA with a single, constant output symbol.

Harrison enumerated DFAs according to the three equivalence classes above. The restricted definition used for these enumerations does not include final or initial states. Example 5 emphasises the disparity between these equivalence classes and the domain of the regular languages associated with n-state DFAs.

Example 5 *The two binary DFAs below are isomorphic when the initial states are ignored. This is clear, as the states of DFA A_2 can be renumbered such that the two DFAs are identical. The renumbering of A_2 requires q_0 to be labelled q_1 and q_1 to be labelled q_0. However, the two DFAs with start states and final states as indicated below accept different regular languages. A_1 is associated with the regular language $(a + ba + bb)^*b$ and A_2 is associated with the regular language $\epsilon + (b + a)(a + ba + bb)^*b$. Therefore, from the perspective of the domain of the regular languages associated with n-state DFAs, these DFAs should not be considered isomorphic.*



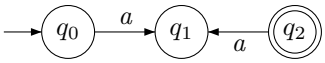
A_1 A_2

□

A finite automaton without initial or final states, as in Harrison's definition, can be seen as a graph with nodes and edges. This graph is referred to in the definition of *connected* finite automata (Chapter 2, page 5). In addition to his equivalence classes, Harrison enumerated the connected finite automata. This enumeration led to the conclusion that the number of finite automata that are unconnected is negligible. Restricting the DFAs enumerated to connected DFAs is not as good a representation of the domain of the regular languages as restricting the DFAs enumerated to

accessible DFAs. Accessible DFAs have no unreachable states whereas the connected DFAs do have unreachable states. These unreachable states have no influence on the language accepted by the DFA, as can be seen in Example 6.

Example 6 *The DFA below satisfies Harrison's definition of connectivity. The language accepted by this DFA is the empty set, as the final state is unreachable. A reduction in the number of accessible states in a DFA limits the number of regular languages which can be accepted by that DFA [9]. For this reason, the domain of accessible n-state DFAs is closer to the domain of the regular languages associated with n-state DFAs than is the domain of connected n-state DFAs.*

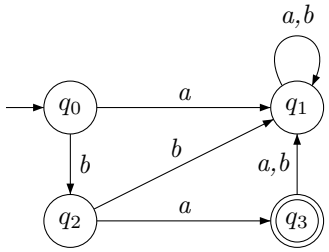


□

Harrison's work was extended by Harary and Palmer [11] to include final states and any number of initial states. Harrison's third equivalence class is the only one considered in [11], but the other two equivalence classes may be handled in a similar manner.

Radke's [22] definition of *sequential* machines agrees with Harrison's restricted definition for finite automata. *Strongly connected* complete sequential machines without start states were enumerated in [22]. Strongly connected machines are defined on page 5 and do not include any machines with inaccessible states. However, the class of strongly connected machines excludes many accessible machines (see Example 7 below). Furthermore, Radke's enumeration included all isomorphic machines with respect to state, input and output permutations. This means that isomorphic machines are counted individually even though such machines are equivalent.

Example 7 *The DFA below is accessible but not strongly connected, as no other state is reachable from state q_1. Note that there is no strongly connected complete machine which recognises the regular language (ba) accepted by this machine.*



□

Example 7 illustrates one of the regular languages which is not associated with any strongly connected DFA. In the context of the regular languages, it is not acceptable to enumerate only strongly connected DFAs as all the finite regular languages are excluded as well as many regular languages with a finite component.

Robinson [23] improved on Radke's enumeration of strongly connected finite automata, by including a start state. Robinson's enumeration of strongly connected DFAs does not include any DFAs which are isomorphic to each other.

Robinson also derived a recursive formula to enumerate accessible DFAs. Unlike strongly connected DFAs [22, 23], for every regular language, there is an accessible DFA which accepts that regular language. In this study, we are only interested in complete accessible DFAs as these fully express the domain of the regular languages [1].

Robinson's enumeration of accessible complete DFAs does not include final states. However, the addition of final states is a trivial extension of the enumeration. This extension of the enumeration may be achieved by the generation of every possible set of final states for each DFA generated without a final state set. There are 2^n different possible final state sets for each n-state DFA. Therefore, the number of different DFAs may be extended to include all possible final state sets by multiplying by 2^n.

In summary, as there is no enumeration of the domain of the regular languages associated with n-state DFAs with more than one alphabet symbol, we looked at other enumerations of DFAs. The domains of these enumerations differed due to restricted definitions of DFAs used. Many of the aforementioned enumerations included examples applying them to the domain of the binary DFAs. We have compiled these numbers into Table 3.1, page 16. The vacant cells in the table were not included in the quoted sources.

Table 3.1, page 16, provides a numeric comparison of the domains for the enumerations discussed previously. All the DFAs in the table are binary. The number of DFAs which are unconnected is negligible according to Harrison [12], as can be seen by comparing row A to row B. The number of strongly connected DFAs with start states, including DFAs which are isomorphic to each other (row D), is far less than the number of strongly connected DFAs excluding start states and including isomorphic DFAs (row C). From the table, it is clear that, for $n \geq 4$, there are more regular languages associated with n-state binary DFAs (row I) than there are strongly connected binary DFAs with final states (row F). There are more accessible DFAs (row E) than strongly connected DFAs (row D). There are always more accessible n-state binary DFAs with final states (row G) than there are regular languages associated with n-state binary DFAs (row I). Finally, row J seems to indicate that accessible DFAs make a fair approximation towards the domain of the regular languages: as n increases, the number of regular languages associated with n-state binary DFAs divided by the number of accessible binary DFAs increases. This means that randomly generating accessible binary DFAs is probably a good approximation of the domain of the regular languages where n is large.

Unlike the enumeration of DFAs, the enumeration of NFAs has not received much attention in the literature [9]. A simple initial estimate can be calculated easily using the different transition functions for n-state NFAs.

Let N be an n-state NFA with m alphabet symbols. Then, for each state q of the n states in N, there are transitions to a subset of the n states on each alphabet symbol a. There are 2^n possible sets of the transitions on alphabet symbol a from state q to be enumerated. To enumerate all possible transition functions on alphabet symbol a, we must enumerate all possible combinations of these transitions for all n states. The formula for this component of the enumeration is

$$2^n \times 2^n \times \ldots \times 2^n$$

[1] The set of accessible DFAs includes all minimal DFAs [9].

	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
A	10	129			
B	9	119			
C	9	296	20958		
D	9	148	3493	106431	3950832
E	12	216	5248	160675	5931540
F	36	1184	55888	3405792	252853248
G	48	1728	83968	5141600	379618560
H	24	1028	56014	3705306	286717796
I	26	1054	57068	3762374	290480170
J	54	61	68	73	77

KEY:
A: Number of pairwise nonisomorphic n-state binary DFAs ignoring start states [12].
B: Number of connected, pairwise nonisomorphic n-state binary DFAs [12].
C: Number of strongly connected n-state binary DFAs including isomorphic DFAs, ignoring start states [22].
D: Number of pairwise nonisomorphic strongly connected n-state binary DFAs [23], including start states.
E: Number of pairwise nonisomorphic accessible n-state binary DFAs [23].
F: D with final states.
G: E with final states.
H: Number of pairwise nonisomorphic minimal n-state binary DFAs [9].
I: Number of regular languages associated with n-state binary DFAs [9].
J: I as a percentage of G.

Table 3.1: Summary of numbers obtained by enumerations of binary DFAs with n states, with a key.

such that 2^n is multiplied by itself n times. The resulting formula for the transitions from all of the n states is $2^{n \times n} = 2^{n^2}$ for alphabet symbol a. We need to generalise this formula to include all of the m alphabet symbols. The transitions possible for each alphabet symbol are not influenced by any other transitions. This implies that the transition from state q on alphabet symbol a does not affect the possible transitions on alphabet symbol b or any other alphabet symbol. This means that the general formula requires 2^{n^2} multiplied by itself m times, that is, $\left(2^{n^2}\right)^m$. Thus the total number of possible transition functions for N is

$$2^{m \times n^2}. \tag{3.1}$$

The total number of different n-state NFAs with m alphabet symbols can be obtained by extending Equation 3.1 to include final states. As we work with a single start state, the final state set is the last step remaining in the enumeration. There are 2^n possible subsets of the n states which can form the final state set. Therefore the total number of different NFAs, including NFAs which are isomorphic to each other, is

$$2^n \times 2^{m \times n^2} = 2^{n + m \times n^2}. \tag{3.2}$$

Example 8 *Assume we want to enumerate all possible unary two-state NFAs. (For the sake of brevity, we will ignore final states in this example.) There should be* $2^{1\times 2^2}$ *transition tables according to Equation 3.1. For state* q_0*, we list transitions to the following* 2^2 *state sets:* $\emptyset, \{q_0\}, \{q_1\}$*, and* $\{q_0, q_1\}$*. For each of the sets of transitions from state* q_0*, we list the* 2^2 *transitions from state* q_1*. The resultant NFAs are as follows:* $N_i = (\{q_0, q_1\}, \{a\}, \delta_i, q_0, -)$ *where* δ_i *is given below, for* $1 \leq i \leq 2^{2^2}$*.*

δ_1	a
q_0	$\emptyset$
q_1	$\emptyset$

δ_2	a
q_0	$\emptyset$
q_1	$\{q_0\}$

δ_3	a
q_0	$\emptyset$
q_1	$\{q_1\}$

δ_4	a
q_0	$\emptyset$
q_1	$\{q_0, q_1\}$

δ_5	a
q_0	$\{q_0\}$
q_1	$\emptyset$

δ_6	a
q_0	$\{q_0\}$
q_1	$\{q_0\}$

δ_7	a
q_0	$\{q_0\}$
q_1	$\{q_1\}$

δ_8	a
q_0	$\{q_0\}$
q_1	$\{q_0, q_1\}$

δ_9	a
q_0	$\{q_1\}$
q_1	$\emptyset$

δ_{10}	a
q_0	$\{q_1\}$
q_1	$\{q_0\}$

δ_{11}	a
q_0	$\{q_1\}$
q_1	$\{q_1\}$

δ_{12}	a
q_0	$\{q_1\}$
q_1	$\{q_0, q_1\}$

δ_{13}	a
q_0	$\{q_0, q_1\}$
q_1	$\emptyset$

δ_{14}	a
q_0	$\{q_0, q_1\}$
q_1	$\{q_0\}$

δ_{15}	a
q_0	$\{q_0, q_1\}$
q_1	$\{q_1\}$

δ_{16}	a
q_0	$\{q_0, q_1\}$
q_1	$\{q_0, q_1\}$

□

This initial enumeration of NFAs includes finite automata which are isomorphic to each other and finite automata which are not accessible. We want to compare the domain of the different NFAs to the domain of the regular languages associated with n-state NFAs. The domain of the different NFAs enumerated above clearly includes all possible regular languages associated with n-state NFAs as the domain includes every possible NFA.

We are also interested in the domain of the accessible NFAs. Now, any regular language associated with an n-state NFA which is not accessible, is also associated with an n-state NFA which is accessible.

Lemma 1 *For any* n*-state NFA,* $N_1 = (Q_1, \Sigma_1, \delta_1, q_0, F_1)$*, with inaccessible states, there is an accessible NFA with* n *states which is associated with the same language.*

Proof: N_2 *is constructed such that it is associated with the same language as* N_1*. The construction follows. Let* Q_i *be the set of inaccessible states. Then, for each state* $q_i \in Q_i$*, set* $\delta_2(q_i, a) = \{\}$*, for all* $a \in \Sigma$*. Set* $F_2 = F_1/Q_i$*. For all accessible states in* N_1*, set* $\delta_2(q_j, a) = \delta_1(q_j, a)$*, where* q_j *is any accessible state and* $a \in \Sigma$*. Set* $\delta_2(q_0, a) = \delta_1(q_0, a) \cup Q_i$*, with* $a \in \Sigma$*. Then* $N_2 = (Q_1, \Sigma_1, \delta_2, q_0, F_2)$*.* N_2 *is associated with the same language as* N_1 *because the previously inaccessible states are not final and there are no transitions from them to any other states so they do not affect the language which is accepted by* N_2*. They did not affect the language accepted by* N_1 *as these states were inaccessible.*

□

Lemma 1 shows that we can work with the domain of the accessible NFAs without excluding any regular languages which are associated with n-state NFAs. In Table 3.2, page 18, the numbers of accessible UNFAs were obtained by experiment. Although Leslie [18] uses the domain of the accessible NFAs for his random generation algorithm, we do not have an explicit enumeration.

Table 3.2 compares the number of different possible NFAs as from Equation 3.2 and the number of accessible NFAs obtained using Grail [10] to the number of different regular languages accepted by n-state $\cup$-UNFAs and $\oplus$-UNFAs (see Chapter 6). Note that there are more regular languages associated with n-state $\oplus$-NFAs than regular languages associated with n-state $\cup$-NFAs. The number of regular languages accepted by n-state $\cup$-UNFAs was obtained from [9]. The number of regular languages accepted by n-state $\oplus$-UNFAs was obtained by experiment, using Grail and a program written to convert $\oplus$-NFAs to DFAs.

We can deduce, from Table 3.2, that there are many more pairwise non-isomorphic n-state UNFAs than there are regular languages associated with n-state UNFAs. This would not present too much of a problem if all the regular languages had the same number of accessible UNFAs associated with them. We know that this is not the case, as all machines with an empty final state set are associated with the empty language $\emptyset$. This means that there are *at least* 256 of the 2048 accessible UNFAs which are associated with one ($\emptyset$) of the 29 regular languages of $\cup$-UNFAs. We look more closely at the number of NFAs associated with specific regular languages in Section 5.4, page 70.

number of ...	$n = 3$	$n = 4$	$n = 5$
different possible UNFAs	4096	1048576	2^{30}
accessible UNFAs, including UNFAs which are isomorphic to each other	2048	622592	763363328
regular languages accepted by n-state $\cup$-UNFAs	29	88	269
regular languages accepted by n-state $\oplus$-UNFAs	54	307	

Table 3.2: Numbers relating to n-state UNFAs for $n = 3, 4$ and 5.

Table 3.2 concludes our discussion of the enumerations of finite automata. We looked at these enumerations in order to determine how the domain of the regular languages associated with the enumerated finite automata compares to the domain of the regular languages accepted by n-state finite automata. In the next section, we will discuss the existing algorithms for the random generation of finite automata.

3.2 Random generation of finite automata

The existing algorithms for the random generation of finite automata generate a set of automata over enumerable domains. Some of the algorithms generate automata over domains discussed in the previous section, while others generate over domains that have not been explicitly enumerated. We will look at these algorithms and the domains of the automata generated by the respective algorithms with reference to the domain of the regular languages. In this chapter, the domains are compared to the domain of the regular languages using a theoretical approach. In Chapter 5, page 56, we investigate the performance of the existing methods of random generation across the domain of the regular languages by experiment.

We will describe the following existing methods for the random generation of finite automata in detail:

- pairwise nonisomorphic generation of UDFAs [20],
- the bitstream method [29],
- random generation of DFA transition tables [29],
- Leslie's NFA generation methods [18] and
- the bijection method [2, 6].

3.2.1 Pairwise nonisomorphic random generation of UDFAs

The random generation of complete, accessible connected UDFAs is trivial. We know from Theorem 2, page 5, that UDFAs may be determined by their set of final states and loop value. Thus, to randomly generate UDFAs, we need to randomly choose a loop value with possible values ranging from zero to $n-1$ and a set of final states. Algorithm 1 builds a random UDFA without final states by connecting states q_0 to q_{n-1} according to the renumbering in Theorem 2 and then randomly generating a loop value. Final state selection for this method will be discussed in more detail in Section 5.2, page 59.

Algorithm 1
Input: *n, the maximum number of states required and randomstream(), a random number stream.*
Output: *UDFA $A = (Q, \Sigma, \delta, q_0, F)$ with F unspecified.*
Method:

1 For i:=0 to $n-2$ do /⋆connect states q_0 to q_{n-1} ⋆/
2 output: "$\delta(q_i, a) \leftarrow q_{i+1}$"
done
3 $x \leftarrow randomstream()\ mod\ n$ /⋆ randomly choose loop value ⋆/
4 output: "$\delta(q_{n-1}, a) \leftarrow q_x$"
End of Algorithm 1

□

Excluding final states, the UDFA is entirely dependant on the loop value. There are n possible loop values, since $\delta(q_{n-1}, a) \leftarrow q_x$ and $0 \leq x \leq n-1$. This means that there are n non-isomorphic accessible UDFAs excluding final states. Note that the DFAs which are isomorphic to each other are equivalent. For every accessible UDFA, there are 2^n possible different combinations of final states. Thus, there are $n \times 2^n$ pairwise non-isomorphic accessible UDFAs. However, there are equivalent DFAs which are not isomorphic to each other. In Table 3.3 we provide the numbers of different accessible UDFAs and the number of regular languages and minimal machines as calculated according to the formulae of Domaratzki *et al.*[9]. These numbers provide a comparison point between the domain of the regular languages and the different accessible UDFAs.

number of	$n=2$	$n=3$	$n=5$	$n=10$
accessible UDFAs with final states	8	24	160	10240
regular languages	6	18	126	8862
reg. lang. as a % of accessible UDFAs	75	75	79	87
minimal UDFAs	4	12	78	4926

number of	$n=50$	$n=100$	$n=300$
reg. lang. as a % of accessible UDFAs	97	99	100
minimal UDFAs as a % of accessible UDFAs	50	50	50

Table 3.3: Numbers relating to n-state UDFAs for selected values of n.

The domain of accessible UDFAs is a fair approximation for the domain of the regular languages for large n. This can be seen from Table 3.3, as for $n = 300$, the percentage of regular languages over accessible UDFAs is 100. However, for small n this is not the case. For $n = 5$, the number of regular languages is 79% of the number of accessible UDFAs. In Chapter 5, we do experiments

using seven and eight state UDFAs. In those cases, the experiments show a distribution which is not uniformly random across the domain of the regular languages.

Example 9 *For $n = 5$, there are 160 pairwise nonisomorphic accessible UDFAs (including final states). Of these 160 UDFAs, 78 are minimal (see Table 3.3). Each of these minimal UDFAs is associated with a unique regular languages. Furthermore, none of these regular languages are associated with five state UDFAs which are not minimal. That means that there are $160 - 78 = 82$ UDFAs which together are associated with the remaining $126 - 78 = 48$ different regular languages. Therefore, the probability of generating one of these $\frac{48}{126} \times 100 = 38$ percent of the regular languages associated five state non-minimal UDFAs is $\frac{82}{160} \times 100 = 51$ percent.*

□

In summary, the pairwise nonisomorphic method of UDFA generation is based on the structure of the connected UDFA. For large n this method of UDFA generation is satisfactory for the generation of UDFAs across the domain of the regular languages. This is undoubtedly the case for $n \geq 300$ because the regular languages associated with 300-state UDFAs form 100 percent of pairwise nonisomorphic accessible UDFAs. For smaller n, we use Domaratzki *et al*'s enumeration [9] to form a random generation algorithm in Chapter 4.

3.2.2 The bitstream method

The bitstream method was first described for $\star$-NFAs by Van Zijl [29], who used the similarity of $\oplus$-NFAs to linear feedback shift registers (LFSRs) to generate random numbers. These random numbers could then be used to randomly generate $\star$-NFAs.

No explicit algorithm was given in [29], but we base Algorithm 2 below on the description in [29]. This algorithm simply fills the transition table from a random bitstream. The result is a grouping of mn^2 bits to form the transition table of an n-state automaton with m alphabet symbols. The set of start states and the set of final states are each chosen according to groups of n bits. Note that when we use this algorithm, we generate an NFA with a single start state. We present the original algorithm here, which has the possibility of generating multiple start states.

The algorithm can be divided into the start state selection, the creation of the transition table and the selection of the final states (see Algorithm 2, below). Lines 1–4 select start states according to the first n bits in the $bitstream_a()$. Lines 5–7 initialise the transition table, such that each transition is empty. Lines 8–12 create the transition table by testing a bit from the bitstream for each existing transition. If the bit is a one, the transition is inserted. Lines 13–16 select final states in the same manner that the start states were chosen.

Algorithm 2
Input: *n, the maximum number of states required,*
$alphabet[]$, the array of alphabet symbols,
m, the number of alphabet symbols, and
$bitstream_a(), bitstream_b()$ and $bitstream_c()$, three random bit-streams.
Output: *$\star$-NFA $N = (Q, \Sigma, \delta, q_{start}, F, \star)$ with the $\star$ operator chosen by the user.*
Method:

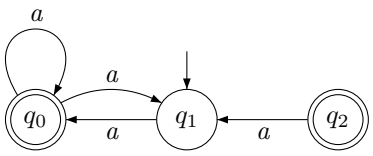
```
1       q_start ← {}
2       For h := 0 to n − 1 do              /* n bits for the start state set */
3           If (bitstream_a() = 1)
4               q_start ← q_start ∪ q_h
            fi
        done
5       For i := 0 to m − 1 do
```

```
6           For j := 0 to n − 1 do
7               δ(q_j, alphabet[i]) ← {}
            done
        done
8       For i := 0 to m − 1 do              /⋆ for all m alphabet symbols ⋆/
9           For j := 0 to n − 1 do          /⋆ for all n states ⋆/
10              For k := 0 to n − 1 do      /⋆ for next n bits ⋆/
11                  If (bitstream_b() = 1)
12                      δ(q_j, alphabet[i]) ← δ(q_j, alphabet[i]) ∪ q_k
                    fi
                done
            done
        done
13      F ← {}
14      For i := 0 to n − 1 do              /⋆ n bits for final state set ⋆/
15          If (bitstream_c() = 1)
16              F ← F ∪ q_i
            fi
        done
End of Algorithm 2
```

□

Example 10 *To randomly generate a unary NFA $N = (Q, \Sigma, \delta, s, F)$ with three states, requires three bitstreams of lengths n, $m \times n^2$ and n respectively. Let $bitstream_a() = \{010\}$, $bitstream_b() = \{110\ 100\ 010\}$ and $bitstream_c() = \{101\}$. The set of start states are determined by $bitstream_a()$. This means that q_1 is the only start state. The set of final states is chosen according to $bitstream_c()$ to be $\{q_0, q_2\}$. Then the resulting NFA is $N = (\{q_0, q_1, q_2\}, \{a\}, \delta, \{q_1\}, \{q_0, q_2\}, \star)$, with δ given below.*

δ	a
q_0	$\{q_0, q_1\}$
q_1	$\{q_0\}$
q_2	$\{q_1\}$



The transition table is composed of bits from $bitstream_b()$ and can also be viewed as δ' supplied below.

δ'	a
0	$\{1, 1, 0\}$
1	$\{1, 0, 0\}$
2	$\{0, 1, 0\}$

Note that this NFA is not accessible*, as state q_2 cannot be reached from the start state.*

□

The importance of connectivity and accessibility in the generated NFAs depends on the purpose of these NFAs. When generating over the domain of the regular languages, accessibility is important.

This is also the case if minimal machines are required. The description of the bitstream method allows for the generation of NFAs with up to n states, thus avoiding any attempts to either connect the machine or discard the disconnected NFAs. If the aim was the generation of accessible n-state NFAs, this method would have to be modified by the user to produce the required output. The bitstream method was used by [6] to examine the size distribution of minimal DFAs equivalent to a random NFA of a given size.

Table 3.2, page 18, compares the number of different UNFAs with the number of different regular languages associated with n-state NFAs. The bitstream method generates any of the possible NFAs, including machines which are isomorphic to each other, with equal probability. This domain was discussed in the previous section.

The bitstream method and the bijection method are the only methods which explicitly describe final state selection. As bits are used to determine which final states are in the final state set, there is a 50 percent chance that any specific state is final. This is effective from a structural perspective, as each of the different final states is present in half of the possible 2^n final state sets.

The regular language associated with a specific NFA is dependent on the transition table *and* the final state set. For this reason, judicious choice of final states will not solve the problem of random generation across the domain of the regular languages. However, the final state set should not be chosen from all possible subsets of the n states with an equal probability. There is only one language, (namely the empty language), associated with NFAs with no final states. Generated according to the bitstream method, the probability of an empty final state set is $\frac{1}{2^n}$. If the number of regular languages associated with n-state NFAs is given as R, then the probability that the final state set is empty should be $\frac{1}{R}$ and not $\frac{1}{2^n}$. Final state selection is a problem for small n. As n tends to infinity, $\frac{1}{2^n}$ tends to zero. Therefore, careful final state selection may be valuable for small n, but becomes insignificant for large n.

Example 11 *Assume we are working with three state union UNFAs. Then we know from Table 3.2, page 18, that three state union UNFAs are associated with* 29 *different regular languages. Therefore the empty final state set should occur with a probability of* $\frac{1}{29}$*. However, with final states chosen according to the bitstream method, the empty final state set occurs with a probability of* $\frac{1}{8}$ *which is more than three times too frequently.*

□

There may be a better way to select the final state set than by choosing equally among the possible subsets of the n states. We look at the selection of final states for the domain of the regular languages in more detail in Chapter 5.

The bitstream method for selecting final states can be used for NFAs as well as DFAs. The bitstream method's transition function generation is not directly applicable to the generation of complete DFAs. However, a method of complete DFA generation with some superficial similarity to the bitstream method is the random transition table method. The bitstream method involves the completion of the transition table of an NFA with bits and generates any possible transition table. Similarly, the random transition table DFA method requires the completion of the transition table with integers less than n and generates any possible DFA transition table.

In summary, the bitstream method of generating NFAs generates any possible NFA. As mentioned in the discussion on the enumeration of NFAs, this would be adequate for the domain of the regular languages if all regular languages were associated with equal numbers of NFAs. We know that this is not the case. For small n, the final state selection according to the bitstream method is not suitable for random generation of NFAs over the domain of the regular languages.

3.2.3 Random DFA transition table generation

To randomly generate complete n-state DFAs, we need to choose a start state, fill the transition table and generate the final state set. The transition table of a DFA has the form $\delta : Q \times \Sigma \rightarrow Q$. In a complete DFA, there is a transition from every state on every alphabet symbol to another state. Assuming that there are n states and m alphabet symbols, there are $n \times m$ transitions.

To randomly generate these transitions, we require $n \times m$ integers obtained from the random number stream *randomstream*. Algorithm 3 generates the transition table values as follows: for each alphabet symbol (line 3) we step through each state (line 4) and assign a random state to the transition value (line 6). Final state selection can be done in the same manner as final state selection for the bitstream method, as in Algorithm 2. For this reason the generation of final states is not shown in Algorithm 3. The start state is chosen randomly as one of the n states (line 1,2).

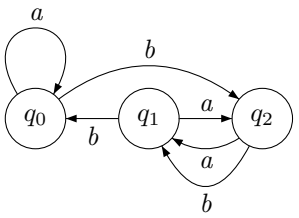
Algorithm 3
Input: *n, the maximum number of states required,*
alphabet[], the array of alphabet symbols,
m, the number of alphabet symbols in array alphabet[], and
randomstream$_a$() and randomstream$_b$(), two streams of random numbers.
Output: *DFA $A = (Q, \Sigma, \delta, q_{start}, F)$ with F selected according to Algorithm 2.*
Method:

```
1       x ← randomstream_a() mod n
2       q_start ← q_x                          /* choose a start state */
3       For i := 0 to m − 1 do                 /* for all m alphabet symbols */
4           For j := 0 to n − 1 do             /* for all n states */
5               x ← randomstream_b() mod n
6               δ(q_j, alphabet[i]) ← q_x
            done
        done
End of Algorithm 3
```

□

Example 12 *Assume we want to generate a three state binary DFA. That means we need six integers from our random number stream to form the transition table. Assume that these six integers modulo n are:* $\{0, 2, 1, 2, 0, 1\}$. *Then the transition table is given by*

δ_1	a	b
q_0	q_0	q_2
q_1	q_2	q_0
q_2	q_1	q_1



.

The start state would be chosen randomly and the final state set would be chosen as for the bitstream method.

□

The method given in Algorithm 3 randomly generates DFAs such that the resultant DFAs can have any transition table. Every transition has n different possible values, as it could be a transition to any *one* state. There is a transition from every state for every alphabet symbol. As previously

mentioned, this means that each DFA requires $n \times m$ random numbers, that is, a random number for each transition. From the number of transitions, the number of different possible DFA transition tables is easily calculated. Each transition is unaffected by the value of any other transition. Therefore, the total number of different transition tables is $n^{n \times m}$.

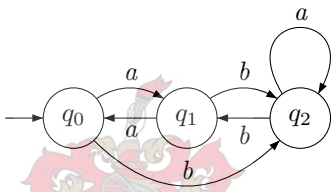
Each transition table has the same probability of occurring as any other transition table. Let K_i be the set of all the finite automata which are isomorphic to finite automaton A_i. If the size of set K_i was the same number for all values of i, then each set would have the same probability of occurring. However, the number of DFAs in the set K_i is not constant. There are $(n-1)!$ different possible numberings of accessible DFAs [23], implying that the size of set K_i is $(n-1)!$ if A_i is an accessible DFA. The size of set K_i for finite automata which are not accessible is not constant.

Example 13 *Let A_1 be a binary DFA with final states ignored, where A_1 is defined by*

$$A_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_1, \{q_0\}, -),$$

with δ_1 given by

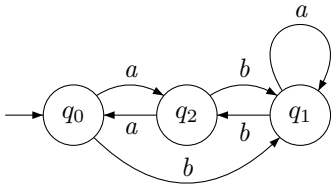
δ_1	a	b
q_0	q_1	q_2
q_1	q_0	q_2
q_2	q_2	q_1



DFA A_1 has three states and is accessible. Therefore, there are $(n-1)! = 2$ different possible numberings of states and matching transition tables. States q_1 and q_2 can be swapped to produce the only other possible numbering of states, as shown in A_2 below. A_1 and A_2 are isomorphic. No further DFAs exist which are isomorphic to A_1 and A_2 without being identical to either one of them.

$A_2 = \{\{q_0, q_1, q_2\}, \{a, b\}, \delta_2, q_0, -\}$ with δ_2 given by

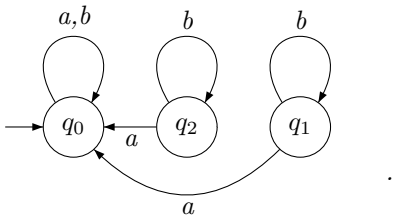
δ_2	a	b
q_0	q_2	q_1
q_1	q_1	q_2
q_2	q_0	q_1



The DFA A_3 (shown below) is not accessible. Therefore, we have no formula to calculate the number of DFAs which are isomorphic to A_3. In this case, there are no DFAs which are isomorphic to A_3 without being identical to A_3. A_3 is connected according to the graph-based definition of connectedness.

$A_3 = \{\{q_0, q_1, q_2\}, \{a, b\}, \delta_3, q_0, -\}$ *with* δ_3 *given by*

δ_3	a	b
q_0	q_0	q_0
q_1	q_0	q_1
q_2	q_0	q_2



.

□

If set K_i, the set of DFAs which are isomorphic to automaton A_i, had a constant number for all values of i, then each possible set K_i would occur with the same probability. Thus, this random generation algorithm would yield DFA distributions equivalent to an algorithm generating DFAs which were pairwise nonisomorphic. As the example above shows, this is not the case. Likewise, if each regular language had the same number of DFAs associated with it, then the regular languages would occur with the same probability. Table 3.4 compares the domain of Algorithm 3 (the number of DFAs), with the number of regular languages associated with n-state binary DFAs. The example below uses Table 3.4 to illustrate the different numbers of DFAs associated with different regular languages.

number of	$n = 2$	$n = 3$	$n = 4$
transition tables	16	729	65536
transition tables with final states	64	5832	1048576
minimal DFAs with n states	24	1028	56014
regular languages	26	1054	57068

Table 3.4: Numbers related to n-state binary DFAs for $n = 2, 3$ and 4.

Example 14 *There are* 56014 *minimal four state binary DFAs. These are all accessible DFAs, because otherwise there would be DFAs which recognised the same regular languages with fewer states. Accessible four state DFAs may be renumbered in* $3! = 6$ *different ways. This means that these regular languages are associated with* $56014 * 6 = 336084$ *different DFAs. Therefore* $\frac{336084}{1048576} * 100 = 32$ *percent of the DFAs generated by this method are minimal. The remaining* 68 *percent of the DFAs generated by this method are not minimal. There are* 1054 *regular languages accepted by four state binary DFAs which are not minimal (Table 3.4, page 25). The* 1054 *regular languages make up approximately two percent of the regular languages associated with four state binary DFAs. Therefore, two percent of the regular languages are accepted by* 68 *percent of the DFAs which are generated by this method.*

□

In summary, this algorithm randomly generates any possible DFA without limiting the domain in any way. This means that DFAs which are not connected or accessible are also generated by this method. As shown in the example above, we expect this method to perform poorly over the domain of the regular languages. We examine Algorithm 3 over the domain of the regular languages in Chapter 5, Section 5.3.

3.2.4 Leslie's NFA generation methods

Leslie [18] used randomly generated NFAs to test the program that he developed to convert NFAs into DFAs using the subset construction. Leslie developed two algorithms for the random generation of NFAs that both require two variables: the number of states and the *density* of the required NFA. The density is a ratio of the number of transitions in an NFA and the maximum number of possible transitions in the NFA.

Algorithm 4, below, goes through every possible transition, inserting that transition into the NFA if a randomly generated integer modulo 100 is less than the input density. Final state selection was not important for the original purpose of this algorithm so it was ignored. All the NFAs generated by these algorithms have a single start state.

Algorithm 4
Input: *n, the maximum number of states required,*
alphabet[], the array of alphabet symbols,
density, the required transition density,
m, the number of alphabet symbols, and
$stream_a()$, a random number stream.
Output: *NFA $N = (Q, \Sigma, \delta, q_{start}, F)$ with the final state selection unspecified.*
Method:

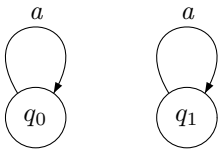
```
1       q_start ← {q_0}                          /* start state */
2       For i := 0 to m − 1 do
3           For j := 0 to n − 1 do               /* for all n states */
4               δ(q_j, alphabet[i]) ← {}
            done
        done
5       For i := 0 to m − 1 do                   /* for all m alphabet symbols */
6           For j := 0 to n − 1 do
7               For k := 0 to n − 1 do           /* transition from state q_j to state q_k */
8                   If (stream_a() mod 100 < density)
9                       δ(q_j, alphabet[i]) ← δ(q_j, alphabet[i]) ∪ q_k
                    fi
                done
            done
        done
End of Algorithm 4
```

□

Algorithm 4, the *all-density* method, is taken from Leslie [18]. Note that the bitstream method is effectively identical to Leslie's algorithm, with a density of 50 percent. This means that Table 3.2, page 18, is also relevant to Algorithm 4, as this algorithm also generates any possible transition table. If the density is chosen at 50 percent, then each transition table has an equal probability of occurring. However, densities other than 50 percent will bias the resultant NFAs. Densities less than 50 percent will result in fewer transitions on average in the NFAs. Conversely, densities greater than 50 percent will result in a higher average number of transitions in the NFAs. A clear example would be an NFA generated with density 100. This would mean that the resulting NFA would have a transition from each state on each alphabet symbol to all possible states.

Example 15 *To randomly generate the transition table of a unary NFA with two states and 30 percent density, four random numbers are required. Let* $stream_a()mod100 = \{17, 53, 34, 3\}$. *Then according to Algorithm 4, there is a transition from state* q_0 *to state* q_0 *as* 17 *is less than* 30. *The transition from state* q_0 *to state* q_1 *is absent, as is the transition from state* q_1 *to state* q_0. *These transitions are omitted because* 54 *and* 34 *are greater than 30. The transition from state* q_1 *to state* q_1 *is included as* 3 *is less than the density of* 30. *The structure of the transition table is given below.*

δ	a
q_0	$\{q_0\}$
q_1	$\{q_1\}$



Note that this NFA is not accessible.

□

As previously mentioned, the set of accessible n-state NFAs accepts all the regular languages which are accepted by the set of n-state NFAs which are not accessible. For this reason, the set of accessible n-state NFAs is a better approximation of the domain of the regular languages accepted by n-state NFAs than the set of all NFAs. (Note that the set of accessible n-state NFAs is still a poor approximation of the domain of the regular languages accepted by n-state NFAs as is illustrated in Chapter 5.) NFAs generated with transition densities greater than 80 percent are likely to be accessible [18].

Despite the advantage of accessibility, there are a limited number of regular languages associated with n-state NFAs with high densities. This can be shown for UNFAs by focusing on the equivalent UDFA obtained by the subset construction. Let n-state UNFA $N = (Q, \{a\}, \delta, Q_0, F)$. The states of the UDFA obtained by the subset construction (page 7) are labelled according to the subsets of Q. To obtain the UDFA, we can start constructing the UDFA with the start state, labelled Q_0. We know that a UDFA may be renumbered such that there is a transition from state q_i to state q_{i+1} for all $i < n - 1$. Therefore, the state which follows the state labelled Q_0 is labelled according the union of the states which can be reached from the set Q_0. Each following state label can be obtained in a similar manner. When the next state label has occurred previously, the UDFA is complete. UNFAs with a high transition density result in UDFAs labelled according to large subsets. With each additional state, there are more transitions to take into account when calculating the label of the next state. The maximum possible size of the label of any UDFA state is the number of accessible states in N. If this maximal label is obtained, each transition in the entire UNFA must be taken into account. Therefore, there must be a self loop in the UDFA at this point, as to have obtained all accessible UNFA states, there must be a transition to each state from at least one accessible state. Thus, the fewer states before a maximal label is obtained, the shorter the UDFA. Note that although the maximal label will not always be obtained, it is probable that it will occur if the original UNFA has a high transition density. Therefore, the equivalent UDFA obtained by the subset construction is likely to have fewer states than an accessible UNFA with a lower density. As the UDFA has fewer states, there are fewer possible regular languages associated with it [9]. (Note that this UDFA is not necessarily minimal.)

We would therefore expect a greater number of regular languages to be associated with accessible NFAs with a moderately low transition density. We will illustrate this empirically for UNFAs with three and four states in Chapter 5. The argument above is based on union UNFAs. In Chapter 6, we examine empirical results for $\oplus$-UNFAs.

The all-density method cannot be used to generate accessible NFAs with low density, as there is no guarantee that NFAs generated with a low density will be accessible. Therefore, to obtain accessible NFAs, the NFAs would either have to be altered to add transitions to the inaccessible states or NFAs with inaccessible states would have to be discarded. Both of these processes would involve time consuming extra steps.

As accessibility is frequently required, Leslie also described an algorithm that generates accessible NFAs. The algorithm first joins all states such that they are accessible and then adds further transitions until the NFA is 'dense enough'. Algorithm 5 is based on Leslie's description of his *connected* method. This definition of connected is equivalent to the definition of accessible in the scope of this thesis.

Leslie's description of the connected or accessible method creates an accessible NFA by marking states as "visited" and generating transitions from "visited" states to unvisited states until all states have been visited. Algorithm 5 marks state i as visited by setting *Vstates[i]* to one. Lines 6–23 create random transitions between visited and unvisited states until all states have been visited. The variable *Unvisited* keeps track of the number of states which are not yet accessible. There are $n - Unvistited$ accessible states. Next, a transition is created from an accessible state to an inaccessible state. The accessible and inaccessible states are randomly chosen (lines 12–13) by the use of a random number modulo $n - Unvisited$ for the accessible state (y) and modulo *Unvisited* for the inaccessible state (z). We step through the *Vstates* array until we have found the y^{th} accessible state and the z^{th} inaccessible state. A transition is added from the y^{th} accessible state to the z^{th} inaccessible state. The value of *Unvisited* is updated and the process begins again. This continues until all states are accessible.

Lines 24–31 add random transitions until the required density is reached. According to Leslie's description, these random transitions are obtained by randomly generating source and destination states and checking if the given transition already exists. If this transition exists, it is not added and the process is repeated. Otherwise, the transition is added.

Algorithm 5
Input: *n, the maximum number of states required,*
alphabet[], the array of alphabet symbols,
m, the number of alphabet symbols,
density, the transition density, and
$stream_a()$ *and* $stream_b()$*, two random number streams.*
Output: *NFA* $N = (Q, \Sigma, \delta, q_{start}, -)$ *with final state selection unspecified.*
Method:

```
1    q_start ← q_0                                  /* start state */
2    For i := 0 to n − 1                            /* initialise variables */
3        Vstates[i] ← 0
4        For i := 0 to m − 1 do
5            δ(q_j, alphabet[i]) ← {}
         done
     done
6    x ← (stream_a() mod (n − 2)) + 1
7    δ(q_0, alphabet[stream_b() mod (m − 1)]) ← q_x
8    Unvisited ← num_states − 2
9     Vstates[0] ← 1                                /* start state visited */
10    Vstates[x] ← 1                                /* transition from start state visited */
11   while Unvisited ≠ 0                            /* while there are inaccessible states */
```

```
12        y ← stream_a() mod (n−Unvisited)   /⋆ number of next accessible state ⋆/
13        z ← stream_a() mod Unvisited       /⋆ number of next inaccessible state ⋆/
14        novis ← 0  nounvis ← 0  i ← 0
15        while ((novis < y) or (nounvis < z)) /⋆ find the accessible & inaccessible states ⋆/
16            if Vstates[i]= 1 then    novis++
                 else    nounvis++
              fi
17            if novis=y then    a ← i       /⋆ accessible state ⋆/
              fi
18            if nounvis=z then    b ← i     /⋆ inaccessible state ⋆/
              fi
19            i ← i + 1
          done
20        c ← stream_b() mod (m − 1)         /⋆ alphabet symbol ⋆/
21        δ(q_a, alphabet[c]) ← δ(q_a, alphabet[c]) ∪ q_b
22        Unvisited ← Unvisited−1
23         Vstates[b]← 1
          done
24      total← n − 1
25      while ((total/(m×n^2)) × 100) < density    /⋆ while not 'dense enough' ⋆/
26        y ← stream_a() mod (n − 1)
27        z ← stream_a() mod (n − 1)
28        a ← stream_b() mod (n − 1)
29        if q_z ∉ δ(q_y, alphabet[a])       /⋆ if transition not already there ⋆/
30            δ(q_y, alphabet[a]) ← δ(q_y, alphabet[a]) ∪ q_z
31            total ← total + 1
          fi
        done
End of Algorithm 5
```

□

The connected method, as described by Leslie, requires that we continue adding transitions until the automaton '*is dense enough*'. Algorithm 5 forces the density up to the required level by calculating the existing density and testing against the required density. One specific density is unlikely to include all possible regular languages associated with n-state NFAs, as can be seen from the number of accessible UNFAs with three, four and five states, given in Table 3.2, page 18. We consider what density should be chosen when attempting to generate NFAs over the domain of the regular languages in Chapter 5, page 74.

De Beijer [4], used a method similar to Leslie's connected method above to randomly produce finite automata to test 'jamming and stretching'. Again final state selection was not of interest to him.

Generation of connected DFAs based on Leslie's connected NFA method

Randomly generating complete accessible DFAs is a potentially good approximation of the domain of the regular languages for large n. We can adapt Leslie's accessible NFA generation method to generate DFAs. An accessible DFA is constructed and then the remaining transitions are randomly generated. As these DFAs are complete, density calculations are not required.

The accessible DFA is created in a similar way to the accessible NFA in Algorithm 5. To generate the accessible DFA, if a transition $\delta(q_i, a)$ already exists, another way must be found to link the inaccessible state to the accessible DFA in the initial connecting phase of the algorithm. Once the accessible structure is created, all the remaining transitions are added randomly to ensure a complete DFA. Algorithm 6 generates accessible DFAs.

Algorithm 6
Input: *n, the maximum number of states required,*
alphabet[], the array of alphabet symbols,
m, the number of alphabet symbols, and
stream$_a$() and stream$_b$(), two random number streams.
Output: *DFA* $A = (Q, \Sigma, \delta, q_0, -)$ *with final state selection unspecified.*
Method:

```
1     q0 ← 0                                          /* start state */
2     For i := 0 to n − 1                             /* initialise variables */
3         Vstates[i]← 0
4         For i := 0 to m − 1 do
5             δ(qj, alphabet[i]) ← {}
          done
      done
6     x ← (stream_a() mod (n − 1)) + 1                /* transition from start state */
7     δ(q0, alphabet[stream_b() mod (m − 1)]) ← qx
8     Unvisited← num_states − 2                       /* start state & qx visited */
9      Vstates[0]← 1
10     Vstates[x]← 1
11    while Unvisited≠ 0
12        a ← −1 b ← −1
13        y ← stream_a() mod (n−Unvisited) + 1        /* number of accessible state */
14        z ← stream_a() mod (Unvisited) + 1          /* number of inaccessible state */
15        novis ← 0 nounvis ← 0 i ← 0
16        while ((novis < y) or (nounvis < z))
17            if  Vstates[i]= 1 then
18                novis++
              else
19                nounvis++
              fi
20            if (novis=y) and (a=-1) then
21                a ← i                               /* accessible state */
              fi
22            if (nounvis=z) and (b=-1) then
23                b ← i                               /* inaccessible state */
```

```
                fi
24              i ← i + 1
            done
25          c ← stream_b() mod (m − 1)
26          if δ(q_a, alphabet[c]) = {} then
27              δ(q_a, alphabet[c]) ← q_b
28              Unvisited←Unvisited−1
29               Vstates[b]← 1
            fi
        done
30      For i := 0 to m − 1 do
i 31        For j := 0 to n − 1 do
32              if δ(q_j, alphabet[i]) = {} then      /⋆ randomly add missing transitions ⋆/
33              x ← stream_b() mod n
34              δ(q_j, alphabet[i]) ← q_x
            done
        done
End of Algorithm 6
```

□

Unlike Algorithm 3, which generates any possible transition table, Algorithm 6 generates accessible DFAs, including isomorphic DFAs. The number of accessible DFAs is discussed by Robinson [23]. Robinson states that there are $(n-1)!$ different numberings of any accessible DFA and therefore every accessible DFA has an equal number of DFAs which are isomorphic to it. This means that all accessible DFAs should be generated with the same probability. Thus this method is an adequate method of randomly generating accessible DFAs. The accessible DFAs may be a fair approximation of the domain of the regular languages associated with n-state DFAs for large n (see Table 3.1, page 16).

The time taken to randomly generate the DFAs is the weak point of the algorithm. Transitions which already occur could be repeatedly attempted to add inaccessible states.

In summary, this DFA generation method randomly generates any accessible DFA. According to the number of elements in each domain, the domain of the accessible DFAs appears to be a good approximation of the domain of the regular languages associated with n-state DFAs, where n is large.

3.2.5 The bijection method to randomly generated binary DFAs

The bijection method was developed by Nicaud [21] for complete accessible binary DFAs and extended by Champarnaud and Paranthoën [7] to randomly generate complete accessible DFAs with m alphabet symbols. As the method first described in Nicaud's doctoral dissertation [21] is complex to implement, Bassino *et. al* [3] describe two equivalent methods of obtaining DFAs. In this section we briefly describe the method based on [21], with examples. The theorems and lemmas are taken directly from [2].

The set of DFAs generated by the bijection method are pairwise nonisomorphic. Let S be the set of DFAs which are isomorphic to DFA D_1 and different to D_1. In order to facilitate the generation

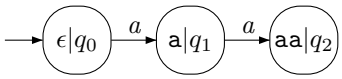
of binary DFAs which are pairwise nonisomorphic, the algorithm must randomly generate only D_1 from the set S. Assume that D_1 and D_2, two different DFAs which are both in set S, were both generated. The algorithm would not be producing pairwise nonisomorphic DFAs as D_1 and D_2 would be isomorphic. The states must be uniquely identifiable to avoid generating D_1 and D_2 from set S. To this end, [2] defined a simple path in a DFA as a word u such that the path labelled by u does not go through the same state twice. He defined w as a map from Q to A^*, for a complete accessible DFA D and A^* representing a string composed of alphabet symbols from alphabet A. For all $q \in Q$,

$$w(q) = \min_{lex}\{u \in A | q_0 \cdot u = q \text{ and } u \text{ is a simple path in } D\}.$$

The simple path u is chosen as the lexicographically least path to state q. P(D) is defined as the set of lexicographically least simple paths to all $q \in Q$. A set $X \in \Sigma^*$ is a prefix set if it contains all the prefixes of all the words in X, however, X={} is not a prefix set. An example of a prefix set is $X = \{\epsilon, a, ab, b, bb\}$. The set P(D) can be shown to be a prefix set for the DFA $D = (Q, A, \delta, q_0, F)$ (see [2]). Randomly generating DFAs such that states and transitions are labelled according P(D) will result in the generation of pairwise nonisomorphic DFAs, as the states will always be numbered in the same way. Example 16 gives some examples of the prefix set P(D_i) for specified DFAs D_i.

Example 16 *As the number of DFAs with the different final state sets can simply be calculated by multiplying the number of DFAs without final state sets by the 2^n different final state sets, we ignore the different final state sets in this example. Let $n = 3$. Then there are different numbers of DFAs associated with the different prefix sets. There are five different prefix sets possible for n=3. The transitions not dictated by the prefix sets can have any combination of the possible values. Therefore the number of DFAs associated with a set P(D_i) are the product of the number of possible transitions on each alphabet symbol.*

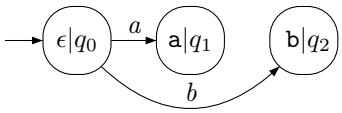
1. *P(D_1)=$\{\epsilon, a, aa\}$*



D_1

The incomplete DFA D_1 represents the set of DFAs which can be labelled according to this prefix set. The possible complete DFAs can have transitions as follows:
the transition from q_0 (ϵ) on a `b` *can be to any of the three states without altering the prefix set;*
the transition from q_1 (`a`) *on a* `b` *can also be to any of the three states and*
the transitions from state q_2 (`aa`) *on an* `a` *and a* `b` *can be to any of the three states.*
There are, therefore, $3 \times 3 \times 9 = 81$ different DFA structures that can be labelled according to this prefix set.

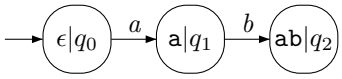
2. *P(D_2)=*$\{\epsilon, a, b\}$



D_2

For the incomplete DFA D_2 to be correctly labelled, the transitions must be carefully chosen. The possible transitions from state q_1 (state a*) are transitions to state q_0 or state q_1 on an* a *or a* b*. A transition from state q_1 to state q_2 on an* a *or a* b *would result in a lexicographically least simple path to state q_2 of* aa *or* ab *respectively. The transitions from state q_2 can be to any of the three states on either of the alphabet symbols without influencing the lexicographical order. Thus, there are $4 \times 9 = 36$ different possible DFA structures that can be labelled according to this prefix set.*

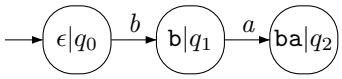
3. *P(D_3)=*$\{\epsilon, a, ab\}$



D_3

For the incomplete DFA D_3 the transitions must be carefully chosen to preserve the prefix set. The transition from state q_0 on a b *can be to any of the three states. The transition from state q_1 on an* a *can be to state q_0 or state q_1. If there is a transition from state q_1 to state q_2 on an* a*, then q_2 should be labelled* aa *and the prefix set is the same as case 1. The transitions from state q_2 can be to any of the three states on either of the alphabet symbols without influencing the lexicographical order. Thus, there are $3 \times 2 \times 9 = 54$ different possible DFA structures that can be labelled according to this prefix set.*

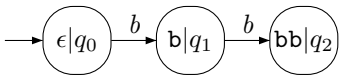
4. *P(D_4)=*$\{\epsilon, b, ba\}$



D_4

Incomplete DFA D_4 can be completed as follows: the transition from state q_0 on an a *must be a self loop as otherwise the prefix set changes;*
the transition from state q_1 on a b *can be to any of the three states; and*
the transitions from state q_2 on both alphabet symbols can be to any of the three states.
There are $3 \times 9 = 27$ different DFA structures which have this prefix set.

5. *P(D_5)=*$\{\epsilon, b, bb\}$



D_5

The incomplete DFA D_5 can be completed as follows:
the transition from q_0 on an a *must be a loop;*
the transition from state q_1 on an a *can go to state q_0 or state q_1 and*
the transitions from state q_2 on both alphabet symbols can go to any of the three states.
There are, therefore, $2 \times 9 = 18$ different DFA structures associated with this prefix set.

□

As the example above shows, there are different numbers of DFA structures associated with the different prefix sets. We know from [23] that there are 216 pairwise nonisomorphic binary DFAs. The possible DFAs mentioned above represent the prefix sets of all 216 DFAs, as $81 + 36 + 54 + 27 + 18 = 216$. To generate uniformly random binary DFAs, the prefix sets must not be chosen with an equal probability. Furthermore, the prefix sets themselves must be able to be generated for any n. The prefix sets can be associated with a tuple defined by [2] as $\mathcal{K}_n$, a subset of $[1, n]^n$:

$$\mathcal{K}_n = \{(x_1, \ldots x_i, \ldots, x_n) \in [1, n]^n | \text{for all } i \in [2, n], x_i \geq 1 \text{ and } x_i \geq x_{i-1}\}.$$

Nicaud defined $\mathcal{E}_n$ as the set of all possible prefix sets with n elements. He then obtained a bijection between $\mathcal{E}_n$ and $\mathcal{K}_n$. To explain the bijection a few other definitions are required. For $E \in \mathcal{E}_n$ with alphabet A, let F_E be the set

$$\mathrm{F}_E = \{u\alpha \in A^* | u \in E, \alpha \in A \text{ and } u\alpha \notin E\}.$$

Then let $\tilde{\mathrm{F}}_E$ be F_E sorted into lexicographical order, that is $\tilde{\mathrm{F}}_E = (f_1, f_2, \ldots, f_{n+1})$, with $f_i < f_{i+1}$. Furthermore, let $|f_i|_E$ be the number of elements in prefix set E which are smaller than f_i.

Lemma 2 *For any $E \in \mathcal{E}$, with $n \geq 1$, if $\tilde{F}_E = (f_1, \ldots, f_{n+1})$ then $|f_{n+1}| = n$.*

Then φ is the bijection from $\mathcal{E}_n$ to $\mathcal{K}_n$, defined by

$$\varphi(E) = (|f_1|_E, \ldots, |f_n|_E),$$

with $f_1, \ldots f_n$ in $\tilde{\mathrm{F}}_E = (f_1, f_2, \ldots, f_{n+1})$. Let

$$||(x_1 \ldots, x_n)|| = \prod_{i=1}^{n} x_i.$$

Theorem 4 *For any non-empty prefix set $E \in A^*$, there are exactly $n||\varphi(E)||$ DFA structures so that $\mathcal{P}(D) = E$ [2].*

Example 17 *For $n = 3$ and $E = \{\epsilon, a, ab\}$ (Example 16, Case 3), $\tilde{F}_E = \{aa, aba, abb, b\}$ and $\varphi(E) = (2, 3, 3)$. Therefore, according to Theorem 4, there are $n \times 2 \times 3 \times 3 = 54$ DFA structures. This can also be seen from the graphical representation in Example 16, together with the possible values of the missing transitions. Table 3.5, on page 35, gives the complete list of $\varphi(E)$ for each $E \in \mathcal{P}(D)$.*

The total number of DFA structures for a specific value of n is $n \sum_{\mathcal{K} \in \mathcal{K}_n} ||\mathcal{K}||$. This can be seen from Table 3.5. The $\mathcal{K}_n$ tuples representing $\varphi(E)$, in the Table 3.5, are all the possible $\mathcal{K}_3$ tuples according to the definition of $\mathcal{K}_n$. As previously mentioned, the total, which can also be calculated as $n \sum_{\mathcal{K} \in \mathcal{K}_n} ||\mathcal{K}||$, is 216.

To randomly generate the pairwise non-isomorphic DFAs the following steps can be followed:

Prefix set	$\tilde{\mathrm{F}}_E$	$\varphi(E)$	$n\|\|\varphi(E)\|\|$
$\{\epsilon, a, aa\}$	$\{aaa, aab, ab, b\}$	$(3,3,3)$	81
$\{\epsilon, a, ab\}$	$\{aa, aba, abb, b\}$	$(2,3,3)$	54
$\{\epsilon, a, b\}$	$\{aa, ab, ba, bb\}$	$(2,2,3)$	36
$\{\epsilon, b, ba\}$	$\{a, baa, bab, bb\}$	$(1,3,3)$	27
$\{\epsilon, b, bb\}$	$\{a, ba, bba, bbb\}$	$(1,2,3)$	18

Table 3.5: Prefix sets for $n = 3$ including values $\tilde{\mathrm{F}}_E$, $\varphi(E)$ and $n||\varphi(E)||$, of which the latter is the number of DFA structures.

1. the $\mathcal{K}_n$ tuples are randomly generated,
2. the associated prefix set is obtained through standard transformations and the DFA with states and transitions that are labelled according to the prefix set is constructed,
3. the remaining transitions are added such that the DFA retains its original prefix set,
4. the states are labelled in prefix order, and
5. finally, state q_0 is selected as the start state and a final state set is randomly chosen.

To randomly generate the $\mathcal{K}_n$ tuples such that a good distribution is obtained, the number of different tuples must be taken into account. Hence $\mathcal{K}_n$ is generalised to $\mathcal{K}_{n,k}$ and the number of tuples and thus possible DFAs associated with the tuples is taken into consideration by the calculation of

$$\mathcal{K}_{n,k} = \{(x_1, \ldots x_i, \ldots, x_n) \in [1,k]^n | \text{for all } i \in [2,n], x_i \geq 1 \text{ and } x_i \geq x_{i-1}\}.$$

From the definition of $\mathcal{K}_{n,k}$, it is clear that $\mathcal{K}_n = \mathcal{K}_{n,n}$. Furthermore, for $n \geq 1$, $k_{n,k} = \sum_{\mathcal{K} \in \mathcal{K}_{n,k}} ||\mathcal{K}||$.

The recursive definition of $k_{n,k}$, below, is taken directly from [2]. For all $n, k \geq 1$ one has

$$\begin{cases} k_{n,k} = 0 & \text{if } k < n \\ k_{n,k} = \frac{1}{2}k(k+1) & \text{if } n = 1 \\ k_{n,k} = k_{n,k-1} + kk_{n-1,k} & \end{cases}$$

Example 18 *The recursive definition can easily be used to calculate the values of $k_{n.k}$. The numeric value of $k_{2,3}$ can be calculated as follows:*

$= k_{2,2} + 3 \times k_{1,3}$
$= k_{2,1} + 2 \times k_{1,2} + 3 \times \frac{1}{2}3(3+1)$
$= 0 + 2 \times \frac{1}{2} \times 2(3) + 18$
$= 24$

Furthermore, we know that $k_{n,k} = \sum_{\mathcal{K} \in \mathcal{K}_{n,k}} ||\mathcal{K}||$. So $k_{n,k}$ can also be calculated as the sum of $||\mathcal{K}||$ for all $\mathcal{K}_{n,k}$. The elements of $\mathcal{K}_{2,3}$ are as follows: $(1,2), (1,3), (2,2), (2,3), (3,3)$. Therefore, the sum can be calculated as follows:

$$2 + 3 + 2 + 2 \times 3 + 3 \times 3 = 24.$$

□

Table 3.6 gives the values of $k_{n,k}$ for $n \leq 4$ and $k \leq 4$ as required for the random generation of elements of $\mathcal{K}_{n,k}$. Algorithm 7 is taken from [2] to randomly generate an element of $\mathcal{K}_{n,k}$. For the purposes of generation of DFAs, this algorithm can be used with $k = n$ to generate tuples which map to prefix sets with n elements.

$n\backslash k$	$k=1$	$k=2$	$k=3$	$k=4$
$n=1$	1	3	6	10
$n=2$	0	6	24	64
$n=3$	0	0	72	328
$n=4$	0	0	0	1312

Table 3.6: Values of $k_{n,k}$ for $n \leq 4$ and $k \leq 4$.

Algorithm 7
Input: *n, the number of states required,*
k, a variable to calculate $\mathcal{K}_{n,k}$*,*
arrayT, containing the values of $k_{n,k}$ *and*
randomstream(), a random number stream.
Output: *A random* $\mathcal{K}_{n,k}$*.*
Method:

1 *Random-of-K(n,k)*
2 *if* $(k < n)$ *return* $\emptyset$ $\quad$ $/\star \mathcal{K}_{n,k}$ *has no value for* $k < n \star /$
3 *if (n=1)*
4 $r \leftarrow (randomstream()\ mod\ T[1][k]) + 1$
5 $x \leftarrow 1$
6 *while* $(r > x)$
7 $r \leftarrow r - x$
8 $x \leftarrow x + 1$
9 *return* x
else
10 $r \leftarrow (randomstream()\ mod\ T[n][k]) + 1$
11 *if* $(r \leq T[n][k-1])$ *return Random-of-K(n,*$k-1$*)*
12 *else return ADD(Random-of-K(*$n-1$*,k),k)*
End of Algorithm 7

□

Example 19 *To randomly generate a three state complete binary DFA, Algorithm Random-of-K(3,3) is used to generate an element of* $\mathcal{K}_n$*.*

1 *Random-of-K(3,3)*
2 $k = 3 = n$
3 $n \neq 1$
10 $r = (randomstream()\ mod\ (T[3][3] = 72)) + 1 = 72$
11 $r = 72 > T[3][2] = 0$
12 *return ADD(Random-of-K(2,3),***3***)*
1 *Random-of-K(2,3)*
2 $k = 3 > n = 2$
3 $n \neq 1$
10 $r = (randomstream()\ mod\ (T[2][3] = 24)) + 1 = 14$
11 $r = 24 > T[2][2] = 6$
12 *return ADD(Random-of-K(1,3),***3***)*

1 *Random-of-K(1,3)*
2 $k = 3 > n = 1$
3 $n = 1$
4 $r = (randomstream()\ mod\ (T[1][3] = 6)) + 1 = 4$
5 $x = 1$
6 $4 > 1$
7 $r = 4 - 1 = 3$
8 $x = 2$
6 $3 > 2$
7 $3 - 2 = 1$
8 $x = 3$
9 *return* **3**

Therefore, the randomly generated element of $\mathcal{K}_n$ *is (3,3,3)*

□

When the element of $\mathcal{K}_{n,n}$ has been obtained, the prefix set can be obtained from the standard transformation.

From $\mathcal{K}_n$ to a binary DFA

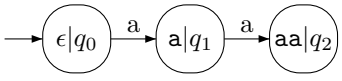
For $n = 3$, a small n value, we did standard in-order binary tree traversals to obtain all the prefix sets and then associated them with a $\mathcal{K}_n$ tuple by calculating the value of $\varphi(E)$ for each prefix set. This could be done as a preprocessing step. For larger n, this would not be possible.

Once the correct prefix set has been obtained, it can be viewed as a binary tree with ϵ as the root and edges labelled `a` and `b`. The states are labelled according to the prefix order of the binary tree. This forms a skeleton of the final DFA. The missing transitions must still be added. If these transitions are added randomly, then the states are no longer uniquely numbered, as a shorter simpler path to a state can form. Therefore, [2] completes the binary DFA by generating, for any missing transition from state i in the automaton, a random transition in the interval $[0, i]$.

Example 20 *We implemented the bijection method as detailed in this section. In an experiment plotting the DFAs generated across the domain for the regular languages, we obtained Figure 3.1, page 38. (See Chapter 5, page 58, for the experimental methods.) It is obvious from the figure that a number of regular languages were not associated with any of these 10000 binary DFAs. There were* 423 *of* 1054 *regular languages that were not associated with any of these randomly generated binary DFAs.*

□

The example above shows that the method as we implemented it has a flaw. The problem lies with the completion of the transition table. To illustrate the problem, we consider $\mathcal{K}_n = (3, 3, 3)$. This is mapped to the prefix set $\{\epsilon, a, aa\}$. The skeleton of the resulting DFA is given below.



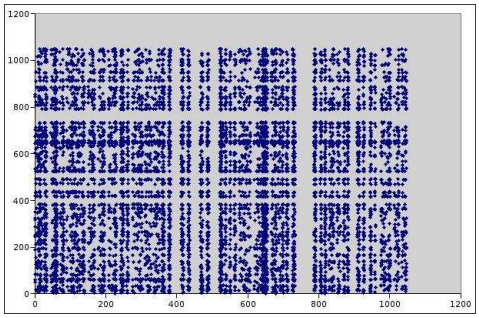


Figure 3.1: 10000 three state binary DFAs randomly generated according to the bijection method, plotted across the domain of the regular languages.

There is a missing transition from state q_0 on a `b` in the DFA above. If we generate a transition between $[0, i]$, the result must be a self-loop on q_0. That would not be a problem if any of the other prefix sets included the DFAs with transitions from q_0 to q_1 on an `a` and a `b`. However, a study of Example 16 indicates that there will be no three state binary DFAs generated such that there are transitions from state q_0 to state q_1 on an `a` and a `b` when transitions are randomly chosen in the interval $[0, i]$.

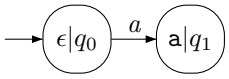
The preprint with this completion of the automata has been withdrawn. An elegant method to obtain a DFA from a tuple is described in [3]. Note that while we describe this step in the context of the generation of binary DFAs, it is described in [3] for m alphabet symbols. The method ensures a random choice between the possible values of the transitions not in the binary tree associated with the tuple. A stack is required to keep track of the current point in the structure. The DFA structure is created from the start state. The state labels are pushed onto the stack in reverse lexicographic order, starting with `b` at the bottom of the stack and `a` at the top of the stack. The tuple $\mathcal{K}_n$ is used with element $x_{n+1} = n$ as the final element of the tuple. Indices i and j are used to keep track of the position in the tuple. Index i has values in the range $[1, n+1]$ and j is in $[1, n]$. Initially $i = j = 1$.

While $j < x_i$, the element is in the covering tree. This means that a new state must be added, labelled according to the stack top. The item from the top of the stack is popped and the labels of the transitions from the new state added. If `a` was popped then `ab` can be pushed, followed by `aa`. The index j is then incremented.

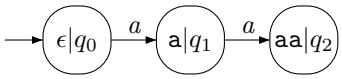
If $j \geq x_i$ then a new state cannot be added and the transition is randomly chosen from the existing states. The index i is then incremented. This process continues while there are still items on the stack.

Example 21 *We will create two binary DFAs from $\mathcal{K}_n$ tuples in this example. First, we consider $\mathcal{K}_n = (3, 3, 3, 3)$. For the purposes of this method, we add x_{n+1} to the tuple.*

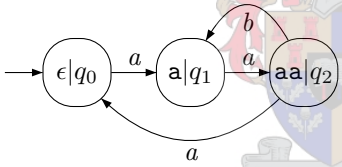
- *The indexes $i = j = 1$: the stack is initialised with* `b`*, followed by* `a` *on top of the stack. As $j < x_i$, a new node is created. The item on the top of the stack is removed. The two new state labels which could arise from the new state are pushed onto the stack in reverse lexicographical order. The stack from top to bottom now contains* `aa, ab, b`*. The index j is incremented.*



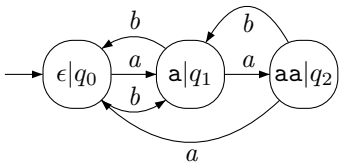
- *The index $j = 2$: as $j < x_i = 3$, a new node is created. The item* `aa` *on the top of the stack is removed. The two new state labels which could arise from the new state are pushed onto the stack in reverse lexicographical order. The stack from top to bottom now contains* `aaa, aab, ab, b`*. The index j is incremented.*



- *The index $j = 3$: as $j \geq x_i = 3$, the transition is randomly generated according to the label on the stack top to any of the existing states. The item* `aaa` *on the top of the stack is removed. The index i is incremented. As $j \geq x_i = 3$ for the rest of the example, all remaining transitions are selected between the existing states. The label is then popped from the stack and i incremented. After the next two transitions have been added, the stack from top to bottom contains* `ab, b`*.*

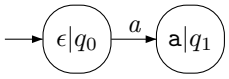


- *The index $i = 3$: as $j \geq x_i = 3$, the transition is randomly generated between the existing states. The item* `ab` *on the top of the stack is removed. The index i is incremented. The remaining item on the stack is* `b`*. The final transition is added from state q_0 on a* `b`*. The stack is empty and the DFA transitions complete. The final state selection would be randomly chosen to complete the construction.*

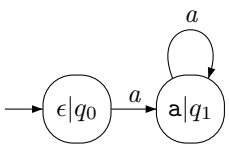


The result of this method generates any of the possible binary DFAs with prefix set $\{\epsilon, a, aa\}$ For the next example we will use $\mathcal{K}_n = (2, 2, 3, 3)$.

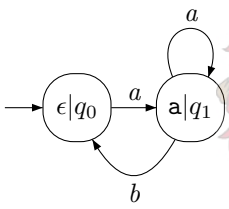
- *The indexes are initialised as $i = j = 1$: the stack is initialise with* `b`*, followed by* `a` *on top of the stack. As $j = 1 < x_i = 2$, a new node is created. The item on the top of the stack is removed. The two new state labels which could arise from the new state are pushed onto the stack in reverse lexicographical order. The stack from top to bottom now contains* `aa, ab, b`*. The index j is incremented.*



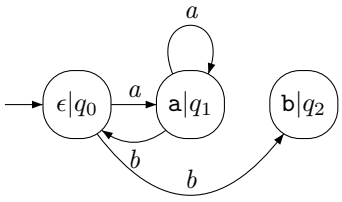
- *The index $j = 2$: as $j = 2 \geq x_i = 2$, a new node is not created. The transition from state q_1 on an* `a` *is generated to either q_0 or q_1. The item* `aa` *on the top of the stack is removed. The stack from top to bottom now contains* `ab, b`*. The index i is incremented.*



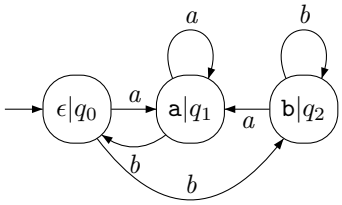
- *The index $i = 2$: as $j = 2 \geq x_i = 2$, the transition is randomly generated according to the label on the stack top to any of the existing states. The item* `ab` *on the top of the stack is removed. The index i is incremented, so that $i = 3$. The stack now contains* `b`*.*



- *The index $i = 3$: as $j = 2 < x_3 = 3$, a new state is created. The label* `b` *is popped from the stack. A transition is created from q_0 to the new state, labelled* `b`*. Then* `bb` *is pushed onto the stack, followed by* `ba`*. The index j is incremented.*



- *The index $j = 3$: as $j = 3 \geq x_3 = 3$, a new state is not created. The label* `ba` *is popped from the stack. A transition is created from q_2 to any of the existing states, on an* `a`*. The index i is incremented. The transition from state q_2 to any existing state on a* `b` *is then added. The stack is empty after this transition has been added, and the DFA transitions are completed. Final states would be selected to complete the construction.*



□

In Chapter 5, the experiments involving the bijection method where modified as above to ensure the correct completion of the transition table. To complete the DFA, state q_0 becomes the start state and a subset of Q is chosen as the final state set.

In summary, the bijection method randomly generated binary DFAs which are pairwise nonisomorphic. Table 3.1, page 16, indicates that binary pairwise nonisomorphic DFAs are a good approximation for the domain of the regular languages associated binary DFAs. In Chapter 5, we demonstrate experimentally that this is currently the best method to randomly generate binary DFAs for small n over the domain of the regular languages.

3.3 Conclusion

In this chapter, we discussed various enumerations of automata and random automata generation algorithms. Although we have seen that it is only required for small n, the algorithm for the random generation of UDFAs over the domain of the regular languages is discussed in the next chapter. The domain of the regular languages associated with binary DFAs is satisfactorily approximated by the bijection method. The generation of UNFAs across the domain of the regular languages is still unsolved. However, we study the existing NFA generation methods over the domain of the regular languages associated with union-UNFAs and $\oplus$-UNFAs in Chapter 5 and Chapter 6, respectively.

Chapter 4

Random generation of UDFAs over the domain of the RLs

In this chapter, we present a method to randomly generate a sequence of n-state UDFAs over the domain of the regular languages. The enumeration of the number of regular languages associated with UDFAs based on the structural properties of the UDFA [9] simplifies the random generation of UDFAs across the domain of the regular languages. We map a given random number to a specific regular language.

A regular language can be described either by a DFA or by a regular expression. There can be multiple regular expressions, as well as multiple DFAs, which represent any given unary regular language, although there is always a unique minimal DFA associated with any given regular language. We use the enumeration of pairwise non-equivalent UDFAs to obtain a random generation algorithm in Section 4.1. Furthermore, in the case of equivalent UDFAs, we generate any one of the possible pairwise non-isomorphic different UDFAs with equal probability. Methods to generate minimal UDFAs across the domain of the regular languages will be discussed in Section 4.2.

Before proceeding with the enumeration of non-equivalent UDFAs, we need to introduce some terminology. Let $g_{\mathcal{L}}(n)$ be the total number of distinct non-equivalent n-state UDFAs and $f_m(n)$ be the number of n-state minimal UDFAs. If a regular language $\mathcal{L}$ is accepted by an n-state UDFA, then there exists a unique minimal UDFA with up to n states which accepts $\mathcal{L}$. However, if $\mathcal{L}$ is accepted by a minimal UDFA, it will also be accepted by UDFAs with more than n states [9]. As there is a distinct language associated with each minimal n-state UDFA, the total number of distinct languages accepted by n-state UDFAs will be equal to the sum of $f_m(i)$, for i ranging from one to n [9]. That is,

$$g_{\mathcal{L}}(n) = \sum_{i=1}^{n} f_m(i).$$

4.1 The enumeration of non-equivalent UDFAs

To enumerate only non-equivalent UDFAs, we must establish when the UDFAs are equivalent. We recall from Theorem 2, page 5 that a UDFA may be renumbered such that the UDFA is defined only by a set of final states and a loop value k [20]. Lemma 3 on page 44, establishes that two UDFAs with n states can only be equivalent if they have the same set of final states.

Lemma 3 *Let M_1 and M_2 be two n-state UDFAs with final state sets F_1 and F_2 and loop values k_1 and k_2, respectively. If $F_1 \neq F_2$, then M_1 and M_2 cannot be equivalent.*
Proof: By contradiction. Assume that M_1 and M_2 are two equivalent n-state UDFAs with final state sets F_1 and F_2 respectively, such that $F_1 \neq F_2$. Then there must be some $q_i \in F_1$ with $q_i \notin F_2$, where $q_i \in \{q_0, q_1, \ldots, q_{n-1}\}$. Hence the string a^i is accepted by M_1 and not by M_2. This is a contradiction an and therefore the result holds.

□

It follows from Lemma 3 above that equivalent n-state UDFAs must have the same final state sets. If the UDFAs are pairwise non-isomorphic but equivalent, they must have different loop values. Example 22, below, lists the regular expressions associated with the six state UDFAs that have final state sets $\{q_2, q_4\}$ and loop values from zero to five. The loop values two and four result in UDFAs which are equivalent.

Example 22 *Let M_i be a six state UDFA with $F_i = \{q_2, q_4\}$. These UDFAs have the same final state set, but are pairwise nonisomorphic different UDFAs. The table below lists the regular expressions associated with M_i, where $\delta(q_{n-1}, a) = q_i$.*

i	*Regular expression associated with UDFA M_i*
0	$aa(aaaaaa)^* + aaaa(aaaaaa)^*$
1	$aa(aaaaa)^* + aaaa(aaaaa)^*$
2	$aa(aa)^*$
3	$aa + aaaa(aaa)^*$
4	$aa(aa)^*$
5	$aa + aaaa$

The term $a^2(a^y)^*$, for $0 \leq y \leq 6$, only occurs when state q_2 is final. Furthermore, M_2 is equivalent to M_4.

□

Equivalent UDFAs have a common final state set and possibly different loop values. The loops of two UDFAs M_1 and M_2 with final state sets F_1 and F_2 and loop values k_1 and k_2 are equivalent when UDFAs M_1' and M_2', constructed by removing the tail states of the UDFAs, are equivalent. The tail states of M_1 are removed by constructing M_1' as follows: $Q' = \{q_k, \ldots, q_{n-1}\}$, $F_1' = F_1 \backslash \{q_0, \ldots, q_{k-1}\}$, start state q_k and $\delta_1'(q, a) = \delta_1(q, a)$. M_2' is constructed from M_2 in a similar manner. To determine when UDFAs are equivalent, we need to know when different loop values result in equivalent loops.

To make it simpler to establish the equivalence of loops, we form a binary tokenized string for each loop [9], according to the finality of the states in that loop. This binary string w has zero in position i if q_{k+i} is a non-final state, where k is the loop value, and one otherwise. If there is no repeating pattern in the string w then it is a *primitive* string. More formally, a primitive string is a string which cannot be written in the form $w = u^e$, where u is a nonempty substring of w, and $e > 1$. Primitive words play an important part in determining whether two loops are equivalent or not (see Lemma 4).

Lemma 4 *Let M_1 and M_2 be two UDFAs. Then the loop of M_1 is equivalent to the loop of M_2 if and only if the loop tokenizations w_1 of M_1 and w_2 of M_2 can be written as $w_1 = s^p$ and $w_2 = s^t$ for some primitive word s, with $p \geq 1$ and $t \geq 1$.*
Proof: Suppose the loops of M_1 and M_2 are equivalent. Then construct M_1' and M_2' to remove the tail states of M_1 and M_2 respectively. Suppose M_1' consists of n_1 states and M_2' consists of n_2 states. As the loops are equivalent, $\mathcal{L}(M_1') = \mathcal{L}(M_2')$. Any string $a^i \in \mathcal{L}(M_1')$ is also $\in \mathcal{L}(M_2')$, $i \geq 0$. Furthermore, any string $a^i \notin \mathcal{L}(M_1')$ is also $\notin \mathcal{L}(M_2')$, $i \geq 0$. This means that the finality of the states in M_1' and M_2' is the same after an input string a^i, for all $i \geq 0$. Furthermore, as

M_1' and M_2' have common finality for all $i \geq 0$, including $i > n_1, n_2$, the finality of states forms a common repeating pattern. To have a common repeating pattern of final states, both UDFAs must have final states in the form $w_1 = s^p$ and $w_2 = s^t$ for some primitive word s, with $p \geq 1$ and $t \geq 1$.

Conversely, assume that binary strings $w_1 = s^t$ and $w_2 = s^p$ with $p, t \geq 1$ and s a primitive word. Construct M_1' and M_2' to eliminate the tail states. Furthermore, construct M_3' to have final states based on primitive word s and loop value zero. A complete connected UDFA only has one transition from each state. This means that, given sufficient input symbols, the states M_3' are revisited in the following order: $q_k, q_{k+1}, \ldots, q_{n-1}, q_k, \ldots$. This forms a repeating pattern which takes the binary form s^p, for all $p \geq 1$. Hence M_1' and M_2' are equivalent to M_3' as w_1 and w_2 are multiples of u. According to the definition of an equivalent loop, if M_1' and M_2' are equivalent M_3', then M_1' and M_2' are also equivalent.

□

Example 23 illustrates two equivalent loops. These loops form part of UDFAs which are not equivalent.

Example 23 *The two loops in the UDFAs in the figure are equivalent and form repetitions of the primitive word* 01*. The UDFAs themselves are not equivalent as they do not have a common final state set. The first loop may be encoded as* $(01)^3$ *and the second as* $(01)^2$*.*

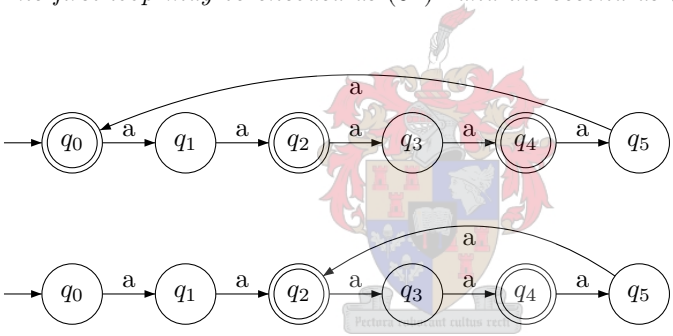


Figure 4.1: *Two non-equivalent UDFAs with six states and equivalent loops.*

□

We know that equivalent loops have tokenized binary words which are multiples of the same primitive word. A loop which cannot be replaced by a shorter equivalent loop is called *minimal*. A loop is minimal if and only if the tokenized binary string is primitive [9, 20]. This condition follows logically from Lemma 4. Example 24 shows tokenized states illustrating a loop which is not minimal.

Example 24 *The six state UDFA in Figure 4.2, page 46, has a loop value of two, since there is a transition from state* q_5 *to* q_2*. This loop may be encoded as* 0101 *which indicates the finality of states* q_i*, where* $2 \leq i \leq n-1$*. This is not a minimal loop, as the pattern* 01 *repeats twice. If the loop value were four, the encoding would simply be* 01 *and hence the loop would be minimal. The reader may note that the final states of the entire UDFA may be tokenized to obtain* 110101*. The encoding of the entire UDFA highlights the regular expression* $\epsilon + a + aaa(aa)^*$ *associated with this UDFA.*

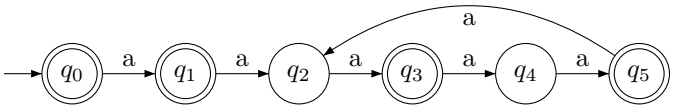


Figure 4.2: *UDFA with six states and loop value two.*

□

We may now state the conditions for the equivalence of n-state UDFAs.

Lemma 5 *Two UDFAs M_1 and M_2 are equivalent if and only if*

- *their loops are equivalent and*
- $F_1 = F_2$.

Proof: Suppose M_1 and M_2 are two equivalent UDFAs with final states sets F_1 and F_2 and loop values k_1 and k_2 respectively. Then, according to the definition of equivalent loops, M_1 and M_2 must have equivalent loops if they are to recognise the same language. Consider any string $a^i \in \mathcal{L}(M_1)$ with $i < n$, and $q_i \in F_1$. As M_1 and M_2, are equivalent $a^i \in \mathcal{L}(M_2)$ and hence $q_i \in F_2$. Furthermore, any string $a^i \notin \mathcal{L}(M_1)$ with $i < n$, indicates that $q_i \notin F_1$. As M_1 and M_2 are equivalent $a^i \notin \mathcal{L}(M_2)$, indicating that $q_i \notin F_2$. Therefore $F_1 = F_2$.

Now assume that $F_1 = F_2$ and the loops of M_1 and M_2 are equivalent. The equivalence of the UDFAs follows from lemma 3 and the definition of equivalent loops.

□

We now know when n-state UDFAs are equivalent. To randomly generate UDFAs over the domain of the regular languages, we want to resolve each integer in the range from one to $g_{\mathcal{L}}(n)$ to a UDFA such that:

- every regular language associated with an n-state UDFA is associated with an integer, and
- no two different integers are mapped to equivalent UDFAs.

To ensure that no two different integers are mapped to equivalent UDFAs, the loops of the two UDFAs must be non-equivalent or the final state set must differ. To ensure non-equivalent loops, we generate only minimal loops.

In order to count the number of minimal loops that could occur in any of the possible n-state UDFAs, we need to have the number of primitive strings of any length consisting of characters 0 and 1. We define the number of primitive words, of length m, as $P(m)$. The formula for the total number of primitive words of length m [8], with an alphabet consisting of zero and one, is

$$P(m) = \sum_{d|m} \mu(\frac{m}{d}) \times 2^d, \tag{4.1}$$

where μ denotes the Möbius function[1] (see Eq. 2.1, page 4).

Nicaud [20] states that there are exactly $\sum_{d|m} \mu(\frac{m}{d}) \times 2^d$ minimal loops of length m. This confirms the use of primitive words in this context. In Example 25, we show how to calculate the number of primitive words of length four and list all such words.

[1]This formula can also be derived using the inverse Möbius function [1].

Example 25 *The number of primitive words of length four consisting of characters* 0 *and* 1 *is given by (see Eq. 4.1):*

$$P(4) = \sum_{d|4} \mu(\frac{4}{d}) \times 2^d$$

$$= \mu(\frac{4}{2}) \times 2^2 + \mu(\frac{4}{4}) \times 2^4$$

$$= -4 + 16 = 12.$$

The list of these twelve primitive words follows below:

0001	0010	0100	1000
0011	0110	1100	1001
0111	1110	1101	1011.

□

If all possible UDFAs with minimal loops are generated without repetition, no equivalent UDFAs will be enumerated. As all possible non-equivalent UDFAs must be enumerated, we must enumerate UDFAs such that all possible combinations of final and non-final states occur outside the loop. For a loop of length t, there are $n - t$ states outside the loop. The enumeration must include all possible sets of these $n-t$ states as final. There are 2^{n-t} sets of the possible final state subsets for the $n - t$ states. Thus the number of non-equivalent UDFAs with n states [9] may be calculated as follows:

$$g_{\mathcal{L}}(n) = \sum_{1 \leq t \leq n} P(t) \times 2^{n-t}. \tag{4.2}$$

These UDFAs are not equivalent because either the final state set or the loop is non-equivalent in every case and all the loops are minimal. This enables us to enumerate all non-equivalent UDFAs.

When there are equivalent but pairwise non-isomorphic UDFAs with n states, we choose to generate any of the possible UDFAs with equal probability. As the pairwise nonisomorphic, pairwise equivalent n-state UDFAs have a common final state set and different loop values, this simply requires a random choice between the possible loop values. We initially generate a UDFA with a minimal loop to ensure that no two different integers are mapped to equivalent UDFAs. Thereafter, we test for equivalent loops and randomly choose between possible loop values.

As we know that equivalent loops are set according to multiples of a primitive word, it is possible to calculate all potential loop values for a given set of final states with minimal loop final states set according to a primitive word v. To calculate the possible loop values, we calculate a binary tokenized string for the entire UDFA. This string w has zero in position i if state q_i is a non-final state and one otherwise. The string w can be written in the form uv^p, with p greater than zero. There are p different loop values which would result in equivalent loops. These loops have lengths which are multiples of $|v|$. To select randomly between these p loop values, we generate a random number r from a separate stream, such that the value of r falls in the interval $[1, p]$. The loop value will be set as $n - (r \times |v|)$.

Example 26 *The six state UDFA in Figure 4.3 has a loop value of four. This loop is minimal and may be encoded as* 01*. However, the entire UDFA may be encoded as* $(01)^3$*. This means that the loop values* $n - 2 = 4$, $n - 4 = 2$ *and* $n - 6 = 0$ *result in equivalent UDFAs.*

□

We are able to use the results above to formulate an algorithm to generate UDFAs over the domain of the regular languages.

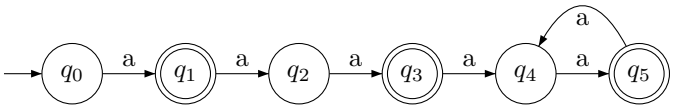


Figure 4.3: *UDFA with six states.*

4.1.1 Random generation algorithm

The general idea behind the method is to match a given random number to one of the possible n-state UDFAs such that no two different integers can be mapped to equivalent UDFAs. For this reason, the first step is to generate a random sequence $S = s_0, s_1, \ldots$ of integers. These integers must fall in the range between zero and $g_{\mathcal{L}}(n)$ as this is the domain of the unary languages.

To match a given random number s_j to a specific UDFA, we must resolve s_j to an exact primitive word v which will be associated with the minimal loop of the eventual UDFA. Following that, we generate an arbitrary combination of final states from states q_0 to $q_{n-|v|-1}$. The last step in the process is the calculation of the length of the loop, as a multiple of $|v|$.

We start by calculating the length of the primitive word v according to the numeric value of s_j. The algorithm to calculate the length of the primitive word, Algorithm 8, is based on Eq. 4.2, page 47.

Algorithm 8
Input: *n, the number of states of the UDFA to be generated, and*
s_j, a random integer.
Output: *$|v|$, the length of the minimal loop, and*
s_{j1}, a number specifying a particular UDFA.
Method:

```
1       s_j1 ← s_j
2       i ← 1
3       while (s_j1 ≥ P(i) × 2^(n−i))
4           s_j1 ← s_j1 − P(i) × 2^(n−i)
5           i ← i + 1
        done
6       |v| = i
7       return s_j1 and |v|
```

End of Algorithm 8

□

Algorithm 8 is based on the number of non-equivalent UDFAs with a minimal loop determined by primitive words of length i, for i ranging from one to the length $|v|$. The number of these non-equivalent UDFAs, $P(i) \times 2^{n-i}$ is subtracted from s_j (line 4). The length of the primitive word is incremented every time this operation is carried out (line 5), until $s_j < P(i) \times 2^{n-i}$ (line 3). When this condition has been reached, we know that s_j falls into the category of UDFAs with finality of states q_{n-i} to q_{n-1} determined by primitive words of length i. Example 27 illustrates the workings of Algorithm 8, with input values $n = 6$ and $s_j = 158$.

Example 27 *Suppose we are randomly generating UDFAs with* 6 *states and random number* s_j *has the value* 158. *Furthermore* $P(1) = 2$, $P(2) = 2$, $P(3) = 6$ *and* $P(4) = 12$.

1 s_{j1} =158
2 $i \leftarrow 1$
3 $P(i) \times 2^{n-i} = 2 \times 2^5 = 64 < s_{j1} = 158$
4 $s_{j1} = 158 - 64 = 94$
5 $i \leftarrow i + 1 = 2$
3 $P(i) \times 2^{n-i} = 2 \times 2^4 = 32 < s_{j1} = 94$
4 $s_{j1} = 94 - 32 = 62$.
5 $i \leftarrow i + 1 = 3$
3 $P(i) \times 2^{n-i} = 6 \times 2^3 = 48 < s_{j1} = 62$
4 $s_{j1} = 62 - 48 = 14$
5 $i \leftarrow i + 1 = 4$
3 $P(i) \times 2^{n-i} = 12 \times 2^2 = 48 > s_{j1} = 14$
7 *return* $s_{j1} = 14$ *and* $|v| = 4$

□

The next step is to calculate which of the primitive words of length $|v|$ to use. Given s_{j1} and the length $|v|$ of the primitive word from Algorithm 8, Algorithm 9 maps s_{j1} to the primitive word v. Algorithm 9 is also based on Eq. 4.2. We obtain the number of UDFAs with finality of states in the minimal loop based on primitive words of length $|v|$ to be $P(|v|) \times 2^{n-|v|}$. The $2^{n-|v|}$ indicates the combinations of finality of the $n - |v|$ states outside the minimal loop.

Algorithm 9
Input: *n, the number of states of the UDFA to be generated,*
$|v|$,*the length of the primitive word and*
s_{j1}, *the output from Algorithm 8.*
Output: *v, a specific primitive word, and*
s_{j2}, *a number specifying a UDFA.*
Method:
1 $t \leftarrow 2^{n-|v|}$
2 $s_{j2} \leftarrow s_{j1}$ *mod* t
3 $i \leftarrow \frac{s_{j1} - s_{j2}}{t}$
4 $v \leftarrow$ *the i-th primitive word*
5 *return* s_{j2} *and* v
End of Algorithm 9

□

To determine which of the $P(|v|)$ primitive words to use, we must match s_{j1} to a particular primitive word. We use a program from the coswizard website [24] to generate primitive words of a given length in an arbitrary order. The order in which primitive words are generated is not important, provided no primitive word occurs twice. We know that there are $2^{n-|v|}$ non-equivalent UDFAs with the finality of the states in the minimal loop determined by a specific primitive word v. Thus we take the remainder of s_{j1} divided by $2^{n-|v|}$ to calculate the value of states $q_0, q_1 \ldots q_{n-|v|-1}$ (line 2). We take the integer part of the division of s_{j1} by $2^{n-|v|}$ to determine the primitive word (lines 2, 3). Example 28 illustrates Algorithm 9.

Example 28 *Suppose we are generating UDFAs with six states and random number s_j has the value* 158*. From Example 27, we know that $s_{j1} = 14$ and $|v| = 4$. Example 25 lists all primitive words of length four. The order in which the primitive words are produced is immaterial provided no primitive word is generated twice.*

1 $t \leftarrow 2^{6-4} = 4$
2 $s_{j2} \leftarrow 14 \ mod \ 4 = 2$
3 $i \leftarrow \frac{14-2}{4} = 3$
4 $v \leftarrow$ *the third primitive word: 0001, 0010, 0100*
5 *return* 2 *and* 0100

□

So far, we have the primitive word associated with the minimal loop and s_{j2} which indicates the finality of states $q_0, q_1 \ldots q_{n-|v|-1}$. The finality of states $q_0, \ldots q_{n-|v|-1}$ is determined according to the s_{j2}-th combination of the binary word of length $n - |v| - 1$. The s_{j2}-th combination of the binary word may be obtained in any manner that maps each integer to a unique combination. Algorithm 10 gives one method to determine this binary word. For t states there are 2^t combinations of zeros and ones to indicate final states. These combinations may be enumerated in such a way that the number of the combination and the position in the string are sufficient to determine the value of the character. See Example 29 for an ordered list of combinations that could represent the finality of three states.

Example 29 *There are $2^3 = 8$ different combinations of finality of states for the set of states $\{q_0, q_1, q_2\}$.*

1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Algorithm 10 determines the combination according to s_{j2} and the position of the character in the string. If $|v| = n$ then all states are specified according to the loop and this algorithm is skipped entirely.

Algorithm 10
Input*: n, the number of states of the UDFA to be generated,*
$|v|$, the length primitive word v, and
s_{j2}, the output from Algorithm 9.
Output*: The finality of states $q_0, \ldots, q_{n-|v|-1}$.*
Method*:*

1 $len \leftarrow 2^{(n-|v|)}$
2 $s_{j3} \leftarrow s_{j2}$
3 $i \leftarrow 0$
4 *while* $(i < (n - |v|))$
5 $\quad$ *if* $(s_{j3} \geq \frac{len}{2})$
6 $\quad\quad b[i] \leftarrow 0$
7 $\quad\quad s_{j3} \leftarrow s_{j3} - \frac{len}{2}$

else

8 $b[i] \leftarrow 1$

fi

9 $len \leftarrow \frac{len}{2}$

10 $i \leftarrow i + 1$

done

11 *Set finality of states* $q_0, \ldots, q_{n-|v|-1}$ *according to* b

End of Algorithm 10

□

Example 30 follows Algorithm 10 step by step and continues from Example 28.

Example 30 *Continuing from Example 28, we have* $s_{j2} = 2$ *and* $v = 0100$.

1 $len \leftarrow 2^2 = 4$

2 $s_{j3} \leftarrow 2$

3 $i \leftarrow 0$

4 $0 < 2$

5 $s_{j3} \geq \frac{4}{2}$

6 $b[0] \leftarrow 1$

7 $s_{j3} \leftarrow 0$

9 $len \leftarrow 2$

10 $i \leftarrow 1$

4 $1 < 2$

5 $s_{j3} < 1$

8 $b[1] \leftarrow 0$

9 $len \leftarrow 1$

10 $i \leftarrow 2$

4 $3 > 2$

11 $b = 10$ *Therefore,* q_0 *is a final state and* q_1 *is not.*

□

The binary tokenized word representing the finality of states is formed by concatenating b from Algorithm 10 and primitive word v from Algorithm 10 to form bv. Then if position i in the string is one, q_i is final, otherwise it is non-final. As the UDFA may be determined by the set of final states and the loop value and we have the final states set from Algorithms 9 and 10, it remains to calculate a loop value.

We generate any of the possible equivalent n-state UDFAs with equal probability. This is done by determining equivalent loops and selecting randomly between them. We take bv, the tokenized binary word, and write it in the form uv^p, where p has the maximum possible value and v is a primitive word. Then p equivalent loops could be formed in this UDFA. A separate stream is used to generate an integer r in the interval $[1, p]$. Then, the loop value is $n - r \times |v|$. This is the final step in the process of randomly generating UDFAs over the domain of the regular languages.

Example 31 *To conclude the generation of a UDFA with six states associated with random number 158, as begun in Example 27, we calculate the loop value and final states of the UDFA. We have* $v = 0100$ *and* $b = 10$*. This may be written as* $10(0100)^1$*. Hence the loop value* k *is* $n - |v| \times 1 = 2$ *(see Figure 4.4). Furthermore, the final states are* $\{q_0, q_3\}$*, as positions zero and three in the string* bv *are ones.*

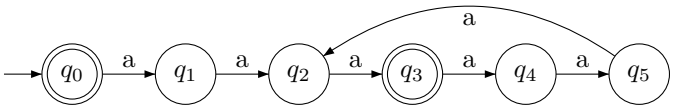


Figure 4.4: *UDFA with six states.*

□

The next section details how this method may be modified to generate only minimal UDFAs.

4.2 Random generation of minimal UDFAs

Under certain circumstances, it may be desirable to generate only minimal UDFAs. In the case of UDFAs, random generation of minimal UDFAs simply requires a few minor alterations to the algorithms discussed above.

We recall Theorem 3, page 6, which lists the three structural conditions for a minimal n-state UDFA A with a loop value k:

1. A is connected,
2. A has a minimal loop, and
3. states q_{k-1} and q_{n-1} do not have the same finality.

All UDFAs we generate are connected so the first condition holds automatically. If we generate loops with final states based on primitive words, then these loops will be minimal and the second condition is met. The third condition states that state q_{n-1} and q_{k-1} must be of opposite finality. This requires a simple adjustment to the algorithms in the previous section. The final states of the minimal loop are determined according to a primitive word calculated in Algorithm 9. Algorithm 10 generates the finality of states q_0 to q_{k-1}. To generate only minimal UDFAs, Algorithm 10 must be adjusted to exclude state q_{k-1}. As we know the finality of state q_{n-1} from Algorithm 9, state q_{k-1} is set such that it has finality opposite to state q_{n-1}.

The enforced opposite finality of states q_{n-1} and q_{k-1} means that there will never be more than one possible loop value because the binary word associated with the entire UDFA can never be written uv^p, for $p > 1$ and primitive word v. There is only one minimal UDFA associated with any one regular expression, and so there cannot be equivalent n-state UDFAs.

Further modifications to Algorithms 8, 9 and 10 are required. The numbers associated with various UDFAs differ because $f_m(n)$ is less that $g_{\mathcal{L}}(n)$. Eq. 4.3 [9] lists the number of minimal UDFAs with n states:

$$f_m(n) = P(n) + \sum_{1 \leq j \leq n} P(n-j) \times 2^{j-1}. \tag{4.3}$$

This equation can be understood as follows: there are $P(n)$ minimal UDFAs with minimal loops of length n. The reason for this is that all these loops are minimal and determined by the $P(n)$ primitive words. Then, for each primitive word of length $n-j$, strictly less than n, there are 2^{j-1} non-equivalent UDFAs with the finality of the loops set according to that primitive word. This is because the state q_{k-1} has a fixed finality, opposite to the finality of state q_{n-1}.

The modifications necessary to alter Algorithm 8 to generate minimal UDFAs only are required because $f_m(n) < g_{\mathcal{L}}(n)$. Algorithm 11 calculates the length of the minimal loop associated with minimal UDFA number s_j. Lines 3 and 4 are altered to match Equation 4.3. The $(i < n)$ incorporates the $P(n)$ UDFAs with loops specified by primitive words of length n.

Algorithm 11
Input: *n, the number of states of the UDFA to be generated, and*
s_j, a random integer.
Output: *$|v|$, the length of the minimal loop, and*
s_{j1}, a number specifying a particular UDFA.
Method:

1 $s_{j1} \leftarrow s_j$
2 $i \leftarrow 1$
3 *while* $(s_{j1} \geq P(i) \times 2^{n-i-1})$ *and* $(i < n)$
4 $s_{j1} \leftarrow s_{j1} - P(i) \times 2^{n-i-1}$
5 $i \leftarrow i + 1$
done
6 $|v| = i$
7 *return* s_{j1} *and* $|v|$
End of Algorithm 11

□

Algorithm 9 also requires only one simple change. Algorithm 12 tests whether the primitive word is of length n or not, and calculates the number of UDFAs with that primitive word accordingly.

Algorithm 12
Input: *n, the number of states of the UDFA to be generated,*
$|v|$,the length of the primitive word and
s_{j1}, the output from Algorithm 8.
Output: *v, a specific primitive word, and*
s_{j2}, a number specifying a UDFA.
Method:

1 *if* $(|v| < n)$ *then*
2 $t \leftarrow 2^{n-|v|-1}$
else
3 $t \leftarrow 1$
fi
4 $s_{j2} \leftarrow s_{j1}$ *mod* t
5 $i \leftarrow \frac{s_{j1}-s_{j2}}{t}$
6 $v \leftarrow$ *the i-th primitive word*
7 *return* s_{j2} *and* v
End of Algorithm 12

□

Algorithm 13 is only used in the case where the primitive word has a length less than $n-1$. Line 1 is the only modified line as Algorithm 13 is only calculating the finality of states $q_0, \ldots, q_{n-|v|-1}$.

Algorithm 13
Input: *n, the number of states of the UDFA to be generated,*
$|v|$, the length primitive word, v, and
s_{j2}, the output from Algorithm 9.
Output: *The finality of states $q_0, \ldots, q_{n-|v|-1}$.*
Method:

1 $len \leftarrow 2^{(n-|v|-1)}$
2 $s_{j3} \leftarrow s_{j2}$
3 $i \leftarrow 0$
4 *while* $(i < (n - |v|))$
5 *if* $(s_{j3} \geq \frac{len}{2})$
6 $b[i] \leftarrow 0$
7 $s_{j3} \leftarrow s_{j3} - \frac{len}{2}$
else
8 $b[i] \leftarrow 1$
fi
9 $len \leftarrow \frac{len}{2}$
10 $i \leftarrow i + 1$
done
11 *Set finality of states $q_0, \ldots, q_{n-|v|-1}$ according to b*
End of Algorithm 13

□

That concludes the modification of the algorithms in Subsection 4.1.1 to generate only minimal UDFAs. Example 32 matches an integer to a minimal UDFA, using Algorithms 11, 12 and 13.

Example 32 *Suppose we are only generating minimal UDFAs. Suppose $n = 6$ and $s_j = 126$. Then Algorithm 11 works as follows:*

1 $s_{j1} \leftarrow 126$
2 $i \leftarrow 1$
3 $(126 > 2 \times 2^{6-1-1} = 32)$ *and* $(1 < 6)$
4 $s_{j1} \leftarrow 94$
5 $i \leftarrow 2$
3 $(94 > 2 \times 2^{6-2-1} = 16)$ *and* $(2 < 6)$
4 $s_{j1} \leftarrow 78$
5 $i \leftarrow 3$
3 $(78 > 6 \times 2^{6-3-1} = 24)$ *and* $(3 < 6)$
4 $s_{j1} \leftarrow 54$
5 $i \leftarrow 4$
3 $(54 > 12 \times 2^{6-4-1} = 24)$ *and* $(4 < 6)$
4 $s_{j1} \leftarrow 30$
5 $i \leftarrow 5$
3 $(30 = 30 \times 2^{6-5-1} = 30)$ *and* $(5 < 6)$

6 $|v| = 5$

7 *Return* $s_{j1} = 30$ *and* $|v| = 5$

That concludes the Algorithm 11.

Algorithm 12 continues below.

1 $t \leftarrow 2^{6-5-1} = 1$

2 $s_{j2} \leftarrow 30 \ mod \ 1 = 0$

3 $i \leftarrow \frac{30-0''}{1}$

4 $v \leftarrow$ *the* $30th$ *primitive word.*

5 *return* 0 *and* 11110

Algorithm 13 is not applicable as state q_0 *is set to the opposite of state* $q_n - 1$*. The only possible loop value for minimal UDFAs is* $n - |v| = 1$*. See Figure 4.5 for the diagram of the six state minimal UDFA we associate with the integer* 126*.*

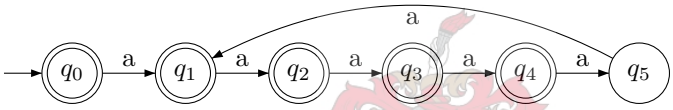


Figure 4.5: *UDFA with six states.*

□

4.3 Conclusion

In this chapter we detailed a simple but effective method of generating UDFAs over the domain of the regular languages. We also showed how this method could be adapted to generate only minimal UDFAs in Section 4.2. We approached the problem with the enumeration of non-equivalent UDFAs.

Chapter 5

Experimental Results

In Chapter 3, we discussed the existing methods for the random generation of finite automata. We compared the domains of the various algorithms to the domain of the regular languages using a theoretical approach. Then, in Chapter 4, we discussed the generation of UDFAs across the domain of the regular languages based on an enumeration of the unary regular languages. In this chapter, we examine the existing algorithms for the random generation of DFAs and UNFAs from an experimental point of view. As final state selection is not handled by most of the existing methods, we will try to select final states for the domain of the regular languages judiciously. Furthermore, in the case of UNFAs, we will look at how the number of start states and the transition density affect the regular languages accepted by the randomly generated n-state UNFAs.

We describe the methods used to carry out the various experiments in Section 5.1. In Section 5.2, the statistical performance of the two UDFA generation algorithms previously described is experimentally examined across the domain of the regular languages. These two algorithms are: pairwise nonisomorphic generation of UDFAs (Algorithm 1, page 19) and random generation of UDFAs across the domain of the regular languages (Chapter 4). In Section 5.3, the statistical performance of the random generation of binary DFA is tested across the domain of the regular languages. We test three algorithms which randomly generate binary DFAs. These three algorithms are: the transition table based method, Algorithm 3 page 23; the accessible method, Algorithm 6 page 30, and the bijection method, described in Section 3.2.5. NFA generation methods are examined across the domain of the regular languages in Section 5.4. The UNFA methods tested are: Leslie's all-density method, Leslie's connected or accessible method and the bitstream method.

5.1 Experimental Methods

Our experimental evaluations of the different methods to randomly generate finite automata all require a common preprocessing element: given a collection of randomly generated finite automata, we map each finite automaton to a unique integer. These integers representing the finite automata are dependent on the domain under examination. Therefore, the elements of the domain must be enumerated and numbered. If no enumeration exists, a list of all the elements in the specific domain must be compiled and numbered. The number of the finite automata in the enumeration or list serves as the integer for that element wherever it occurs in the set of randomly generated finite automata.

The distribution of the stream of integers obtained from preprocessing is then evaluated, using a two-dimensional dotplot [14, 26] as a visual indication of the distributions of the finite automata represented by the integers. These dotplots are obtained by plotting the numbers representing the finite automata as pairs $(U_1, U_2), (U_2, U_3), \ldots (U_{r-1}, U_r)$ for r finite automata. Dotplots with evenly distributed points indicate a uniform random distribution, whereas dotplots with clusters of

points are not uniformly random. It is to be noted that the exact shape of the dotplot depends on the order in which the randomly generated elements are numbered. A cluster of points indicates repetition of an element. If this element had a different number, the cluster of points would have occurred at a different location on the dotplot. The dotplots thus give a visual indication of the statistical performance of the method used to randomly generate finite automata.

We use the dotplots to analyse the distribution of randomly generated finite automata over the domain of the regular languages and over the domain of pairwise nonisomorphic finite automata. To view dotplots over these domains, we must enumerate the domains. Four specific domains are examined in this chapter:

- the domain of pairwise nonisomorphic n-state UDFAs,
- the domain of the regular languages associated with n-state UDFAs,
- the domain of the regular languages associated with n-state binary DFAs, and
- the domain of the regular languages associated with n-state UNFAs.

Two steps are required to produce the stream of integers required for the dotplot. These steps are:

1. a list or an enumeration must be obtained for each of the domains mentioned above and
2. the randomly generated finite automata must then be compared to the elements in the list or enumeration.

In the sections that follow, we discuss the methods used for these two steps for each of the four domains.

5.1.1 Testing UDFAs over the domain of pairwise nonisomorphic UDFAs

We are able to enumerate the domain of pairwise nonisomorphic UDFAs using Theorem 2, page 5. According to Theorem 2, a UDFA can be specified by the loop value and the set of final states. Furthermore, n-state UDFAs which are numbered according to the theorem are isomorphic to each other only if the loop values and the final states sets are the same. Therefore, to enumerate all pairwise nonisomorphic UDFAs, we generate every possible loop value and add to each loop value every possible set of final states. We then number each of these UDFAs according to its position in the enumeration.

The randomly generated UDFAs must be compared to the enumeration of pairwise nonisomorphic UDFAs so that each randomly generated UDFA D_i can be numbered according to the element in the enumeration which is isomorphic to D_i. Each UDFA D_i is compared, by means of Grail's `isomorph` routine [10], to all the enumerated n-state UDFAs T_j, for $0 < j \leq max$, where max is the total number of n-state UDFAs in the domain of pairwise nonisomorphic UDFAs. The number stream required to produce a dotplot is formed by adding integer j of the enumerated UDFA T_j which is isomorphic to D_i to the number stream, as D_i occurs in the randomly generated UDFAs. (Note that Grail's `isomorph` routine does not distinguish between unconnected finite automata in which the unconnected states have different finality. Therefore, this method of generating a random number stream for the domain of pairwise nonisomorphic UDFAs will only work for accessible UDFAs.)

5.1.2 Testing UDFAs over the domain of the regular languages

As previously mentioned, we need to enumerate the domain of the regular languages associated with n-state UDFAs and then compare the randomly generated n-state UDFAs to this domain.

The UDFAs can be enumerated by means of the algorithm described in Chapter 4. This algorithm maps a randomly generated number to a specific regular language and generates an n-state UDFA which is associated with that regular language. Therefore, this algorithm can be used to enumerate all pairwise nonequivalent n-state UDFAs by replacing the randomly generated numbers with sequential numbers running from one to the number of the regular languages associated with n-state UDFAs. As the algorithm associates each number with a different regular language, this will result in a set of pairwise nonequivalent n-state UDFAs associated with all regular languages that are accepted by n-state UDFAs.

The enumerated UDFAs must be compared with the randomly generated UDFAs so that the randomly generated UDFAs can be numbered according to the equivalent UDFAs in the enumeration. Grail does not have a routine that specifies when two DFAs are equivalent. In the case of UDFAs, regular expressions can be used to determine whether two UDFAs are equivalent or not, as Grail's `fmtore` routine produces unique regular expressions for minimal UDFAs. Therefore, we minimise both the enumerated UDFAs and the randomly generated UDFAs. Then two lists of regular expressions are obtained using Grail's `fmtore` routine: a list of the regular expressions associated with the minimised enumerated UDFAs and a list of regular expressions (r) associated with the minimised randomly generated UDFAs. Each of the r_i regular expressions is compared to the list of regular expressions associated with the enumerated UDFAs. When a regular expression is found to be identical to r_i, the number of that regular expression is added to the number stream. The number stream produced in this manner can then be plotted to determine the random generation method's statistical performance over the domain of the regular languages.

5.1.3 Testing binary DFAs over the domain of the regular languages

As previously mentioned, we require an enumeration of the regular languages associated with n-state binary DFAs or a list of these regular languages from which to number the randomly generated n-state binary DFAs. As there is no existing enumeration for the domain of the regular languages associated with n-state binary DFAs, we must obtain a list by experimental means.

To obtain this list, we enumerate all the possible n-state binary DFAs, filtering out any binary DFAs which are not accessible. We need to remove DFAs that are equivalent to other DFAs until the set is pairwise nonequivalent. To establish equivalence, we recall that the regular language accepted by a set of minimal DFAs which are isomorphic to each other is unique among the regular languages accepted by minimal DFAs [9]. This means that no minimal DFA A_1 which is not isomorphic to another minimal DFA A_2 will recognise the same language as A_2. Therefore, we minimise all the binary DFAs using Grail's `fmmin` routine. Then a `bash` script using Grail's `isomorph` routine eliminates DFAs from the set of minimised DFAs until the set is pairwise nonequivalent. As the processing is time consuming, the binary DFAs generated do not have final state sets that are empty. We add one minimal DFA that is associated with the empty language instead of generating any DFAs with empty final state sets as these are all equivalent. The remaining binary DFAs represent the domain of the regular languages associated with binary DFAs.

The randomly generated binary DFAs must be compared to the list to determine the integers representing the equivalent DFAs. We cannot use Grail's `fmtore` routine to obtain comparable regular expressions in this case as this routine does not result in a common regular expression for all equivalent minimal DFAs. Therefore, the comparison must be done by comparing minimised DFAs using the `isomorph` routine. The minimised randomly generated binary DFAs are compared to the minimal binary DFAs representing the domain of the regular languages and numbered accordingly. The time taken to generate the number stream in this manner is considerable.

5.1.4 Testing UNFAs over the domain of the regular languages

There is no enumeration of the domain of the regular languages accepted by n-state UNFAs. A list of the different regular languages could be obtained experimentally. However, Domaratzki *et al.* [9] give a complete listing of the distinct UNFAs which accept all regular languages associated with n-state UNFAs for $1 \leq n \leq 5$ at

$$\texttt{http://www.math.uwaterloo.ca/}\sim\texttt{shallit/papers.html}.$$

We use these lists to generate all the 88 four state UNFAs and 269 five state UNFAs such that no two UNFAs are equivalent. These UNFAs are converted to UDFAs using Grail's `fmdeterm` routine. The resulting UDFAs are minimised using Grail's `fmmin` routine.

The randomly generated UNFAs must be converted to UDFAs and minimised so that they can be directly compared to the minimised UDFAs in the lists. We can then use the `fmtore` routine to obtain regular expressions for all the UDFAs. These regular expressions can be compared to generate the number stream, as with the UDFAs.

5.2 Unary DFAs

In this section, we examine pairwise nonisomorphic random generation of UDFAs as described in Algorithm 1, page 19, and the method of random UDFA generation described in Chapter 4. We test Algorithm 1 to see if the method adequately covers the domain of the regular languages. As final state selection is not included in the method, we first consider a few possible options for final state selection for Algorithm 1. We also illustrate the statistical performance of the method based on the enumeration of the unary regular languages (as described in Chapter 4) over the domain of the regular languages to show the ideal uniform distribution.

5.2.1 Final state selection for Algorithm 1

In this section, we discuss the final state selection to use with Algorithm 1 to generate UDFAs across the domain of the regular languages. The results of these experiments are discussed in Section 5.2.2.

Final state selection is only described by Van Zijl [29] for the bitstream method and Nicaud [2] for the bijection method. These methods of final state selection have the same effect, namely that any possible subset of the n states has an equal probability of forming the final state set. We will focus on the bitstream final state selection as it is easier to implement. As any subset has an equal probability of occurring as the final state set, Algorithm 1 with the bitstream method of final state selection should randomly generate UDFAs uniformly across the domain of pairwise nonisomorphic UDFAs. However, the domain of pairwise nonisomorphic UDFAs is not a good approximation for the domain of the regular languages for small n (see Table 3.3, page 19).

Example 33 *Assume we are working with five state UDFAs. The bitstream method for final state selection has the final state set $F = \emptyset$, occurring with a probability of $\frac{1}{2^5} \times 100 = 3.125$ percent. All UDFAs with $F = \emptyset$ are equivalent and accept the empty language. There are 126 different regular languages accepted by five state UDFAs. Therefore, there should only be $\frac{1}{126} \times 100 = 0.79$ percent of the UDFAs associated with the empty set according to the domain of the regular languages. For a complete, accessible UDFA to be associated with the empty set, the final state set F must be empty. Therefore the number of occurrences of the empty set can be modified based purely on final state selection.*

□

As can be seen from the above example, the bitstream method for final state selection is clearly not effective for randomly generating UDFAs over the domain of the regular languages. We have an enumeration of the domain of the regular languages associated with UDFAs, so we can evaluate ways to improve final state selection for the domain of the regular languages. Note that judicious final state selection is only important for small n, as the probability of UDFAs accepting the empty language (UDFAs with an empty set of final states) occurring is $\frac{1}{2^n}$ which tends to zero as n tends to infinity. This value is altered by careful final state selection, but for large n, the result is already insignificant.

There are two ways in which we can measure the distribution of final states across the domain of the regular languages for n-state UDFAs.

1. The total number of final states may vary from zero to n. We can count the number of different regular languages associated with UDFAs with s final states, where $0 \leq s \leq n$.
2. We can also count the number of different regular languages which have state q_t as a final state, for any possible state q_t.

The first point above should help us determine how to select the number of final states. The second point should help us determine which states should be chosen as final.

The number of different regular languages associated with n-state UDFAs with s final states

We can determine how the number of final states is to be chosen by analysing the number of different regular languages associated with n-state UDFAs with s final states.

With the structural knowledge of the UDFA from Chapter 4, we can derive a formula for the number of different regular languages that may be accepted by n-state UDFAs with s final states. According to Lemma 3, page 44, two equivalent n-state UDFAs have the same final state set and equivalent loops. According to Lemma 4, page 44, loops w_1 and w_2 are equivalent if w_1 and w_2 can be written as $w_1 = s^p$ and $w_2 = s^t$ for some primitive word s, with $p \geq 1$ and $t \geq 1$. To avoid counting any one language more than once, we only count loops with final states that can be represented by a primitive binary tokenised string. The s final states can all be inside the loop, all outside the loop or a combination of the two.

For s final states outside the loop, the minimal loop will have a loop value of $n - 1$. Thus, there are $n - 1$ states outside that loop and any combination of these states can be final. Therefore, there are C_s^{n-1} finite regular languages associated n-state UDFAs with s final states outside the loop.

We can also calculate the number of regular languages where all s final states are within the loop[1]. To do this, let $P_i(j)$ be the number of primitive words of length j with i ones. Then $P_s(n)$, the number of primitive words of length n with s ones, counts the number of different regular languages with all s final states in a loop, with loop value zero.

Some of the final states can be in the loop and others outside the loop. The $P_i(j)$ component counts final states in the loop and the C_{s-j}^{n-j} counts final states outside the loop, such that the total number of final states is always s. Then for s final states, there are

$$C_s^{n-1} + \Sigma_{i=1}^{s} \Sigma_{j=i}^{n+i-s} (P_i(j) \times C_{s-i}^{n-j}) \tag{5.1}$$

regular languages associated with n-state UDFAs.

[1]The derivation of this formula falls outside the scope of this thesis.

Number of final states	0	1	2	3	4	5	6	7
Number of different RLs	1	34	121	213	213	121	34	1

Number of final states	0	1	2	3	4	5	6	7	8
Number of different RLs	1	43	179	390	490	390	179	43	1

Table 5.1: The number of different regular languages associated with UDFAs with s final states for $n = 7$ and $n = 8$.

Table 5.1, page 61, lists the number of different regular languages associated with UDFAs with s final states for $n = 7$ and $n = 8$. When generating UDFAs with $n = 7$ or $n = 8$ states, we want to simulate these frequencies. We can use a uniform random number generator to simulate the desired frequency of final states as described in Example 34, on page 61.

Example 34 *Using a uniform random number generator which generates deviates between zero and one, we are able to simulate the desired frequency of final states. This is done by calculating the number of regular languages accepted by n-state UDFAs with s final states as a proportion of the total number of regular languages associated with n-state UDFAs. As shown in Table 5.2, for $n = 7$, a random deviate, U_i, in the interval $[0, \frac{1}{738})$ results in the generation of a UDFA with 0 final states. The probability of U_i falling in this interval is consistent with the probability of a regular language being associated with an n-state UDFA with zero final states. This method of choosing the number of final states according to the interval of U_i will result in the desired frequency of final states.*

Number of final states, s	Frequency	Proportion	Interval
0	1	$\frac{1}{738}$	$[0, \frac{1}{738})$
1	34	$\frac{35}{738}$	$[\frac{1}{738}, \frac{35}{738})$
2	121	$\frac{156}{738}$	$[\frac{35}{738}, \frac{156}{738})$
3	213	$\frac{369}{738}$	$[\frac{156}{738}, \frac{369}{738})$
4	213	$\frac{582}{738}$	$[\frac{369}{738}, \frac{582}{738})$
5	121	$\frac{703}{738}$	$[\frac{582}{738}, \frac{703}{738})$
6	34	$\frac{737}{738}$	$[\frac{703}{738}, \frac{737}{738})$
7	1	$\frac{738}{738}$	$[\frac{737}{738}, \frac{738}{738})$

Table 5.2: The number of regular languages with s final states as a proportion of the total number of regular languages accepted by seven state UDFAs and the interval in which a random deviate U_i must fall to generate a UDFA with s final states.

□

An equation can also be derived to calculate the number of regular languages associated with minimal UDFAs with s final states for a specific n, based on Theorem 3, page 6. UDFAs are minimal when the final states in the loop form primitive words and states q_{k-1} and q_{n-1} have opposite finality.

Each term of Equation 5.1, page 60, must be modified to take the opposite finality of q_{k-1} and q_{n-1} into account. The final states can be all outside the loop, all inside the loop or a combination of the two. Assuming all final states are outside the loop, the loop value for the minimal loop is then $n - 1$. As all final states are outside the loop, state q_{n-1} is not final. Therefore, to have opposite finality state q_{n-2} must be final. The remaining states can be any combination of final and non-final states, providing there are $s-1$ final states. This is indicated in the equation as C_{s-1}^{n-2}. For all

s states within the loop, we look at a loop value of zero. The number of UDFAs with a loop value of zero is the number of primitive words of length n with s ones $P_s(n)$. This is because there is no state q_{k-1} as $k = 0$. The case where some final states are in the loop and some out, can be further divided into loops where q_{n-1} is a final state and loops where q_{n-1} is not a final state. There are $\frac{P_i(j)}{j} \times i$ primitive words ending with one, indicating state q_{n-1} is final. If q_{n-1} is final, state q_{k-1} cannot be final for the UDFA to be minimal. Hence the combination of final and non-final states outside the loop can be calculated as C_{s-i}^{n-j-1}. The number of primitive words ending in zero is $\frac{P_i(j)}{j} \times (j - i)$. Therefore, state q_{k-1} must be final. The combinations of final and non-final states possible outside the loop are thus C_{s-i-1}^{n-j-1} as one final state is fixed and one position has a set value.

Then for s final states, there are

$$C_{s-1}^{n-2} + \Sigma_{i=1}^{s} \Sigma_{j=i}^{n+i-s} \left((\frac{P_i(j)}{j} \times i \times C_{s-i}^{n-j-1}) + (\frac{P_i(j)}{j} \times (j-i) \times C_{s-i-1}^{n-j-1}) \right) + P_s(n) \qquad (5.2)$$

regular languages associated with n-state UDFAs.

Table 5.3 shows the numbers of minimal seven and eight state UDFAs as they correspond to s final states. The values in the table are calculated according to Equation 5.2.

Number of final states, s	0	1	2	3	4	5	6	7
Number of minimal UDFAs	0	14	68	134	134	68	14	0

Number of final states, s	0	1	2	3	4	5	6	7	8
Number of minimal UDFAs	0	16	90	232	302	232	90	16	0

Table 5.3: The number of different regular languages associated with minimal UDFAs with s final states for $n = 7$ and $n = 8$.

Example 35 *The method used in Example 34, page 61, can also be used to generate random numbers with an appropriate distribution for minimal UDFAs with s final states. Table 5.4 shows how this can be done for minimal seven state UDFAs associated with s final states.*

Final states	Frequency	Proportion	Interval
0	0	$\frac{0}{432}$	$[0, \frac{0}{432})$
1	14	$\frac{14}{432}$	$[\frac{0}{432}, \frac{14}{432})$
2	68	$\frac{82}{432}$	$[\frac{14}{432}, \frac{82}{432})$
3	134	$\frac{216}{432}$	$[\frac{82}{432}, \frac{216}{432})$
4	134	$\frac{350}{432}$	$[\frac{216}{432}, \frac{350}{432})$
5	68	$\frac{418}{432}$	$[\frac{350}{432}, \frac{418}{432})$
6	14	$\frac{432}{432}$	$[\frac{418}{432}, \frac{432}{432})$
7	0	$\frac{432}{432}$	$(\frac{432}{432}, \frac{432}{432})$

Table 5.4: The number of minimal seven state UDFAs with s final states as a proportion of the total and the interval in which a random deviate U_i must fall to generate a UDFA with s final states.

□

The number of different regular languages which have state q_t as a final state, where t is less than n.

As demonstrated above, we can simulate the frequency of the number of final states. However, after the number of final states is determined, the selection of the elements in the final state set is still not clear. In the case of UDFAs, we can count the number of different regular languages associated with each specific final state q_t as there is a unique state numbering as indicated by Theorem 2, page 5.

The number of different regular languages which have state q_t as a final state may be calculated according to the structure of the UDFA:

Lemma 6 *Let κ be the total number of regular languages associated with n-state UDFAs. Let $M_i = (Q, \Sigma, \delta, q_0, F)$ be a set of pairwise non-equivalent n-state UDFAs which accept all κ regular languages. Furthermore, assume that these UDFAs are numbered according to Theorem 2. Choose an arbitrary $q \in Q$. Then there are exactly $\frac{\kappa}{2}$ of the M_i UDFAs which have state q as a final state.*
Proof:
We know, from Chapter 4, that a set of n-state UDFAs which accepts the κ regular languages must include all possible primitive words to specify the finality of the states in the loop. The states outside the loop must consist of all combinations of final and non-final states.

Assume without loss of generality that the loop has size $n - k$. There are κ^k possible combinations of final states for the k states outside the loop. In the UDFAs where q is one of these k states, then it will occur as a final state in 2^{k-1} combinations [27].

However, for some of the UDFAs in the set, state q will be in the loop. We know that the finality of the states in the loop is determined by a primitive word. We remember that if a word is primitive, then all rotations of that word are also primitive. This means that, for a primitive word that has the same number of ones as zeros, it holds that each state in the loop will be final the same number of times as it is not final, taking all rotations of the primitive word into account.

For a primitive word that does not have an equal number of zeros and ones, we take the inverse of the primitive word, obtained by exchanging all zeros for ones and all ones for zeros into account. When considering all rotations of both the primitive word and its inverse, each state in the loop will be final in the same number of DFAs as it is non-final. As this holds for all possible loops, state q_t is in the final state set for a total of $\frac{\kappa}{2}$ of the n-state UDFAs, for $0 \leq q_t \leq n - 1$.

□

Lemma 6 proves that for every final state q_t, the number of different regular languages associated with n-state UDFAs in which $q_t \in F$ adds up to half of the total number of regular languages associated with n-state UDFAs. This means that all final states have the same number of regular languages associated with them. Therefore, all states should be equally likely to be final. Thus, when we have the number of states which are to be final, we choose the actual final states with equal probability.

5.2.2 UDFA experimental results

In this section, we do a number of experiments plotting UDFAs over the domain of the regular languages and over the domain of pairwise nonisomorphic UDFAs and discuss the results of each experiment. By means of these experiments we show that:

- Algorithm 1, with bitstream final state selection, is random over the domain of pairwise nonisomorphic structures;
- Algorithm 1 is not entirely satisfactory over the domain of the regular languages, even with final state selection modelled on Section 5.2.1 and

- the method described in Chapter 4 is shown to be satisfactory over the domain of the regular languages, based on experiments as well as theory.

Experiment 1 *Algorithm 1 with the bitstream final state selection.*

Algorithm 1 is based on a structural enumeration of UDFAs. We expect it to be random over the domain of pairwise nonisomorphic UDFAs. Figure 5.1, page 78, shows that the UDFAs produced by Algorithm 1 appear to be uniformly random over the domain of pairwise nonisomorphic accessible UDFAs. However, the UDFAs produced by this algorithm are not random over the domain of the regular languages as can be seen by the clustering of points in the second graph in Figure 5.1, page 78. This experiment was also done with seven state UDFAs and similar results were obtained.

One of the more obvious problems can be seen in the number generated UDFAs with an empty final state set. Table 5.5 lists the number of UDFAs with s final states produced by this experiment. The table also lists the expected values based on the number of possible final state sets with size s (calculated by taking $\frac{C_s^8}{2^8} \times 10000$) and the desired values based on the regular languages. As can be seen from the table, the final states produced from the experiment are consistent with the values expected from the bitstream method. However, the final states we would expect from the domain of the regular languages differ greatly, especially at the endpoints. We use the method described in Example 34, page 61, to alter the distribution of the different sizes of final state set.

No. fin. states	No. fin. states in in Experiment 1	No. fin. states expected based on bitstream method	No. fin. states expected based on Table 5.1
0	39	39	6
1	317	313	251
2	1050	1094	1043
3	2214	2188	2273
4	2786	2734	2855
5	2167	2188	2273
6	1099	1094	1043
7	294	313	251
8	35	39	6

Table 5.5: Numbers of final states obtained in Experiment 1 and the number of final states expected in 10000 UDFAs.

Experiment 2 *Algorithm 1 with the number of final states selected according to Example 34, page 61.*

The number of final states for every UDFA is chosen according to the method described in Example 34, page 61, in Experiment 2. Table 5.6, page 65, shows that the number of UDFAs with final state sets of each size obtained in the experiment closely match the expected values. This at least limits the occurrences of the empty set and a^*, which is definitely an improvement. Figure 5.2, page 78, still shows a marked clustering of points. These are caused by the regular languages which often have more than one n-state UDFA associated with them but are not associated with minimal n-state UDFAs.

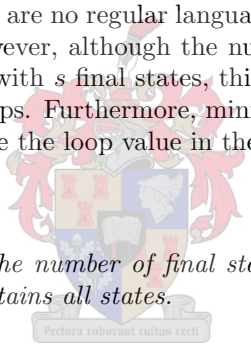
Experiment 3 *Algorithm 1 with the number of final states selected according to Example 35, page 62.*

In Experiment 2, the number of final states for every UDFA is chosen according to the method

No. fin. states	No. fin. states in in Experiment 2	No. fin. states expected based on Table 5.1
0	2	6
1	252	251
2	1014	1043
3	2329	2273
4	2824	2855
5	2262	2273
6	1076	1043
7	240	251
8	3	6

Table 5.6: Numbers of final states obtained in Experiment 2 and the number of final states expected in 10000 UDFAs

described in Example 34, page 61. Experiment 3 follows a similar approach. We base the selection of the number of final states on Example 35, page 62, that is the numbers of minimal UDFAs with s final states. As can be seen in Figure 5.3, page 78, this does reduce the clustering of points closest to the axes. The empty set (the language closest to the axes) is excluded by this method as the choice of the number of final states was based on the distribution of minimal UDFAs. The reason for this exclusion is that there are no regular languages other than the empty set which are associated with no final states. However, although the number of final states was chosen based on the numbers of minimal UDFAs with s final states, this does nothing to ensure that the final states selected will form minimal loops. Furthermore, minimal loops are not solely dependent on final state selection, but must include the loop value in the calculation.



Experiment 4 *Algorithm 1, with the number of final states chosen uniformly, excluding those whose final state set is empty or contains all states.*

Unlike Experiments 2 and 3, final states are chosen uniformly (excluding $F = \emptyset$ and $F = Q$) in this experiment as the biggest problem with bitstream final state selection is the number of empty final state sets and the number of complete final state sets. The exclusion of these two final state sets for two or more alphabet symbols, mostly results in minimal complete accessible DFAs [6]. In the unary case, however, the dotplot is worse than for final states chosen according to Example 35, page 62. Figure 5.4 shows 8000 seven state UDFAs plotted across the domain of the regular languages. Although the pronounced clustering along the axes is absent, more distinct lines form where other regular languages are not minimal. This is also more marked because the remaining possible numbers of final states were chosen uniformly. Although the end points of the bitstream are a problem, in the unary case the number of final states should not be chosen uniformly.

Experiment 5 *Algorithm 1 with bitstream final state selection for* 15*-state UDFAs.*

The influence of final state selection on the dotplots decreases as n increases. Furthermore, according to Table 3.3, page 19, we would expect a larger value of n to result in a graph that has a less obvious clustering of points when plotted across the domain of the regular languages than seen in Figure 5.1, page 78. We generated 20000 fifteen state UDFAs using Algorithm 1, page 19, with the bitstream method for selecting final states. This resulted in Figure 5.5, page 79, which only has a slight clustering of points near the axes. This indicates that, for a large enough n, Algorithm 1 and the bitstream method of final state selection are acceptable over the domain of the regular languages.

Experiment 6 *Method to randomly generate UDFAs described in Chapter 4.*

As can be seen from the previous experiments, for small n, Algorithm 1 is not an effective method of generating UDFAs across the domain of the regular languages. The method described in Chapter 4 is based on the enumeration of the domain of the regular languages. To illustrate the accuracy of this latter method, we generated 8000 seven state UDFAs and 10000 eight state UDFAs using this method. Figure 5.6 shows the uniform distribution of points across the domain of the regular languages obtained by this method.

Table 5.6 shows that Experiment 2 was probably close enough in number to the distribution of regular languages with s final states, but this could not overcome the connection between the final state and the loop. This is because UDFAs have a limited structure. The choice of final states cannot be made independently of the choice of the loop for the domain of the regular languages. The regular languages which cause the most clustering of points are those not associated with minimal UDFAs. Therefore, to generate only minimal UDFAs, according to Theorem 3, page 6, the loop has to be minimal and state q_{k-1} must have opposite finality to state q_{n-1}. Thus to avoid a greater number of UDFAs which are not minimal, the final states must be chosen together with the loop as done in the method described in Chapter 4.

In Section 5.3, we look at binary DFAs which do not have as limited a structure as UDFAs.

Experiment 7 *Method to randomly generate minimal UDFAs described in Chapter 4.*

This method randomly generates UDFAs which are minimal, across the domain of the pairwise nonisomorphic minimal UDFAs. The plot on the left in Figure 5.7, shows a good distribution over the domain of the pairwise nonisomorphic minimal UDFAs. The plot on the right is over the domain of the regular languages accepted by seven state UDFAs. There are 432 pairwise nonisomorphic minimal seven state UDFAs and 738 different regular languages accepted by seven state UDFAs. The gaps in the plot indicate the regular languages which are not associated with seven state UDFAs. Therefore, generating UDFAs across the domain of the pairwise nonisomorphic minimal UDFAs is not a good substitute for random generation across the domain of the regular languages.

5.2.3 Summary of results of UDFA experiments

We ran Algorithm 1 with final states selected:

1. according to the bitstream method,
2. according to the distribution of regular languages associated with UDFAs with s final states
3. according to the distribution of minimal UDFAs with s final states and
4. final states chosen uniformly, excluding those whose final state set is empty or contains all states.

The results showed that although Algorithm 1 with the bitstream method for final state selection is random over the domain of pairwise nonisomorphic UDFAs, for small n, the UDFAs produced are not random across the domain of the regular languages. Furthermore, modifying the final state selection according to the distribution of minimal UDFAs with s final states or according to the distribution of regular languages associated with s final states does not produce a significant improvement. However, even with the exclusion of empty final state sets and complete final state sets, the number of final states should not be chosen uniformly. The clustering of points in the dotplots from Experiments 1– 5 can be clearly seen when compared to the results from the method described in Chapter 4, a method theoretically proven to be random across the domain of the regular languages. However, Experiment 5 and Table 3.3, page 19, clearly show that Algorithm 1

with bitstream final state selection is adequate across the domain of the regular languages for larger n.

5.3 Binary DFAs

In this section, we test the random generation of binary DFAs through transition table generation (Algorithm 3, page 23), the random generation of accessible binary DFAs based on Leslie's accessible method (Algorithm 6, page 30) and the bijection method of DFA generation, page 31. The analysis of random generation methods in Chapter 3 suggests that accessible DFAs are a good approximation for the domain of the regular languages associated with n-state DFAs, where n is large. The bijection method generates pairwise nonisomorphic binary DFAs, while the modified version of Leslie's connected method produces accessible DFAs which include DFAs which are isomorphic to each other. However, accessible DFAs can be renumbered in a set number of ways [23]. This implies that each set of DFAs which are isomorphic to each other have the same probability of occurring. In the experimental results we compare these two latter methods specifically.

For smaller n, final state selection can play a role and this is discussed in Section 5.3.1. The experimental results are discussed in Section 5.3.2 and summarised in Section 5.3.3.

5.3.1 Final state selection for the generation of binary DFAs

As mentioned in the discussion of final state selection for UDFAs, we can count the number of languages where s states are final and the number of languages which have state q_t in the final state set. We can only count the number of languages which have state q_t in the final state set if q_t is uniquely identifiable. The bijection method has states labelled in prefix order of according to the lexicographically least, simple path to each state. This means that DFAs generated according to this method have uniquely identifiable states. However, the method which creates a random transition table and the method based on Leslie's connected method do not have recognisable states beyond the start state. Therefore, information pertaining to the number of languages where state q_t, where $t \neq 0$, is in the final state set can only be applied to the bijection method.

The number of different regular languages associated with n-state binary DFAs with s final states

In order to determine how the size of the final state set should be chosen for the domain of the regular languages, we counted the number of different regular languages associated with three state binary DFAs which have s final states (for $0 \leq s \leq n$). Table 5.7 shows that apart from the empty final state set and the final state set where $F = Q$, the distribution of languages with s final states is even for $n = 3$. There is not enough information for further deductions but to produce an equivalent table for $n = 4$ would be extremely time consuming. Limiting the empty and full final state set is clearly important for small n.

Number of	$\|F\| = 0$	$\|F\| = 1$	$\|F\| = 2$	$\|F\| = 3$
different REs	1	535	535	1
minimal DFAs	0	514	514	0

Table 5.7: The number of regular languages and pairwise nonisomorphic minimal DFAs associated with three state binary DFAs with $|F|$ final states.

The number of different regular languages which have state q_t as a final state, where t is less than n.

To determine the probability with which each state q_t should be final, we look at the different regular languages with state q_t as a final state. For methods other then the bijection method, the only state which is distinguishable is the start state, q_0. Therefore, for these methods, we look at the number of different languages for which q_0 is in the final state set. The bijection method has states which are numbered according to the lexicographically least, simple path. Therefore, the state label can be uniquely determined. We count the number of different regular languages which have q_t in the final state set for three state DFAs.

The number of regular languages which have q_0 in the final state set of the binary DFAs is 527 for $n = 3$ and 28534 for $n = 4$. In both cases, the number indicates that exactly half of the possible different regular languages associated with n-state binary DFAs have q_0 in the final state set. There is one regular language which has only q_0 as a final state, and one with no final states. All the other regular languages have a state other than q_0 as final. This does not mean that states other than the start state should have a higher probability of being final. As q_0 is final for 50 percent of the regular languages, this encourages us to hypothesise that each state should have a 50 percent probability of being a final state. We know that q_0 should have a 50 percent probability of being selected as a final state.

Table 5.8 shows that the number of regular languages associated with minimal DFAs is constant for each state of three state binary DFAs. This confirms the bijection and bitstream methods uniform choice of final states. However, as there is only one language associated with the empty final state set and one associated with the full final state set, these final state sets must be limited.

	state q_0 in F	state q_1 in F	state q_2 in F
Number of regular languages	527	531	536
Number of minimal DFAs	514	514	514

Table 5.8: The number of regular languages and pairwise nonisomorphic minimal DFAs associated with three state binary DFAs with state q_t in the final state set.

5.3.2 Binary DFA experimental results

In this section, we do a few experiments plotting binary DFAs over the domain of the regular languages. The following methods are considered in this section: the transition table method, the accessible method and the bijection method. We look at final state selection where the empty final state set and the final state set such that $F = Q$ are excluded and the bitstream method of final state selection and compare them to each other.

Experiment 8 *Binary DFAs randomly generated using Algorithm 3 (transition table based generation), with final states chosen according to the bitstream method.*

Figure 5.8, page 80, shows binary DFAs randomly generated using Algorithm 3 with final states generated according to the bitstream method plotted across the domain of the regular languages. Algorithm 3 produces DFAs with any possible transition table including inaccessible DFAs. Final states selected according to the bitstream method include a disproportionate number of DFAs with an empty final state set. This problem is aggravated by the disconnected DFAs which can be generated by this method as, if all final states are not accessible, the result is the same as the empty final state set and the DFA is associated with the empty regular language. Furthermore, the regular language associated with a specific DFA with states which are not connected is not influenced by the disconnected states. Therefore, the number of regular languages which can be

associated with the DFA with disconnected states is limited to the number of regular languages associated with the accessible states. As a result of these factors, we expected the poor distribution which is shown in Figure 5.8.

Experiment 9 *Binary DFAs randomly generated using Algorithm 3 (transition table based generation), with final states chosen uniformly, excluding empty final state sets and final state sets where $F = Q$.*

Figure 5.9, page 80, shows three state binary DFAs randomly generated using Algorithm 3 with final states generated uniformly, but such that the empty final state set and $F = Q$ never occur. There is a slight improvement which can be seen by comparing this figure to the previous figure. However, DFAs which are not accessible are not minimal. Randomly generating these DFAs lowers the probability of minimal DFAs occurring. This method is not worthwhile when considering the domain of the regular languages. Despite the improvement it is still clear that, for the 10000 points plotted, the dotplot is poor.

Experiment 10 *Binary DFAs randomly generated using Algorithm 6 (accessible DFAs generated), with final states chosen according to the bitstream method.*

As can be seen from Figure 5.10, page 81, this is an improvement on the transition table based random generation because, as the binary DFAs are accessible, the probability of generating connected DFAs is higher. There are still marked line clusters of points.

Experiment 11 *Binary DFAs randomly generated using Algorithm 6 (accessible DFAs generated), with final states generated uniformly, but such that the empty final state set and $F = Q$ never occur.*

Figure 5.11, page 81, shows improvement as expected with the exclusion of the empty final state set and the final state set where $F = Q$. The method shows potential and could be a viable method to use across the domain of the regular languages for large n. There are significant sparse areas in the dotplot though; as well as a few distinct lines. Although DFAs which are isomorphic to each other are generated by this method, we still expect each set of isomorphic DFAs to occur with the same probability as the accessible n-state DFAs can be renumbered in the same number of ways [23].

Experiment 12 *Binary DFAs randomly generated using the bijection method, (pairwise nonisomorphic accessible DFAs generated), with final states chosen according to the bitstream method.*

Figure 5.12, page 81, exhibits a distribution which is fairly uniform, with exception of a few lines. This is an improvement over all the previous dotplots for binary DFAs. The plot is still sparse though and the final state sets need to be modified.

Experiment 13 *Binary DFAs randomly generated using the bijection method, (pairwise nonisomorphic accessible DFAs generated), with final states generated uniformly, but such that the empty final state set and $F = Q$ never occur.*

Figure 5.13, page 82, still has a few distinct lines. However, the rest of the dotplot is uniform. The performance is better than the accessible method. This can be explained by remembering that the structures and final state sets are generated independently. For the accessible method to generate two different but isomorphic DFAs, DFA A_2 would be a renumbering of DFA A_1. For A_1 and A_2 to be isomorphic, the final state set of A_2 must be able to be renumbered to match the final state set of A_1. For example, for DFA A_1 the state q_1 could be final and equivalent to DFA A_2's final state of q_2. However, for the bijection method, a different final state set will not result in an isomorphic but different DFA. The DFAs generated are either identical and isomorphic or not isomorphic. This results in the best distribution of any existing method across the domain of the regular languages.

5.3.3 Summary of the results for binary DFAs

The bijection method clearly results in the best performance across the domain of the regular languages. This is probably a good method to generate binary DFAs over the domain of the regular languages for large n. For small n, final state selection does not prove helpful in the binary case, although it is important to select final states such that the occurrence of the empty final state set and the final state set such that $F = Q$ are limited.

5.4 Unary NFAs

In this section, we test the bitstream method (Algorithm 2, page 20), Leslie's all-density method (Algorithm 4, page 26) and Leslie's connected method (Algorithm 5, page 28), to see if these methods are adequate over the domain of the regular languages associated with union-UNFAs. Note that all NFAs generated are restricted to a single start state except in Section 5.4.2, where we study the effect of different numbers of start states on the domain of the regular languages and Experiment 20, which gives a visual indication of these effects. The initial study of NFAs in Chapter 3 leads us to expect that these methods will not be adequate over the domain of the regular languages. Therefore, we examine the number of UNFAs associated with each regular language in Section 5.4.1. The variables which may have an influence on the regular language accepted by an UNFA are the start state set, the final state set and the transition density. In Section 5.4.2, we consider the effect of multiple start states when generating over the domain of the regular languages. The choice of final state set is considered in Section 5.4.3 and the effect of transition density on the regular languages accepted by UNFAs is examined in Section 5.4.4.

The $\oplus$-UNFAs are discussed in a similar manner in Chapter 6.

5.4.1 Regular languages associated with n-state UNFAs

We know that the total number of possible UNFAs, with m alphabet symbols and including all UNFAs which are isomorphic to each other, is $2^{m \times n^2}$ from Equation 3.2, page 16. Table 3.2, page 18, gives the number of UNFAs calculated according to this equation and the number of accessible UNFAs. This shows that there are many more n-state UNFAs than regular languages associated with n-state UNFAs. The bitstream method generates UNFAs across the domain of the different possible UNFAs, including all UNFAs which are isomorphic to each other. Leslie's all-density method with a 50 percent density also generates UNFAs across the domain of the different possible UNFAs, including all UNFAs which are isomorphic to each other. If all regular languages were associated with an equal number of UNFAs, we would expect good results for these methods when evaluated across the domain of the regular languages. However, this is not the case, as Section 5.4.5 shows that the statistical performance of the random UNFA generation methods across the domain of the regular languages is not uniformly random and the number of UNFAs per regular language is not constant. To demonstrate, for $n = 4$, there are $2^4 \times 1 \times 2^{4^2} = 1048576$ UNFAs according to Equation 3.2, page 16. Table 5.9, page 71, lists the number of these 1048576 UNFAs associated with specific regular expressions, for the 20 highest numbers of UNFAs associated with specific regular languages.

The UNFAs associated with a^*, aa^* and the empty set alone, account for 80 percent of the 1048576 UNFAs. There are 88 different regular languages associated with four state UNFAs. The 20 regular languages in Table 5.9 are associated with 99.45 percent of the UNFAs! This means that only 0.55 percent of the UNFAs are associated with the other 68 regular languages. That is for every 10000 UNFAs generated, we would expect 55 of them to be associated with the 68 regular languages not represented by regular expressions in Table 5.9. Furthermore, the remaining 68 regular languages also do not have equal numbers of UNFAs associated with them.

There are a total of 88 different regular languages associated with four state UNFAs. Of these 88

regular expression	number of UNFAs	number of accessible UNFAs
a^*	403237	259781
aa^*	277152	204576
empty set	153600	38912
aaa^*	57276	46524
$aaa^* + ""$	51463	37159
empty string	44408	2432
$a + aaaa^*$	11190	8502
$(aa)^*a$	6739	1459
$aa(aa)^* + ""$	6337	1249
$aaaa*$	5760	5760
$aaaa^* + ""$	5286	4134
a	4939	619
$a + ""$	4939	619
$a + aaaa* + ""$	3492	2724
$aa + aaaaa^*$	2292	1716
$aa + aaaaa^* + ""$	1164	972
$aa(aa)^*$	1032	456
$aa + a$	894	318
$a + aa + ""$	894	318
$a + aaaaa*$	732	540

Table 5.9: The total number of four state UNFAs, including UNFAs which are isomorphic to each other and the number of accessible UNFAs, associated with certain regular languages. The regular expressions representing these regular languages are included in the table.

regular languages, 27 are each associated with only six UNFAs each. That results in a total of $27 \times 6 = 162$ UNFAs accepting 27 of the 88 languages. Therefore, the probability of any of these 27 regular languages being generated by algorithms generating across the domain of the possible UNFAs is $\frac{162}{1048576} = 0.00015$.

The 1048576 UNFAs include both accessible and inaccessible UNFAs. Accessible UNFAs with n states are associated with all the regular languages associated with unconnected or inaccessible UNFAs. This can be seen from Lemma 1, page 17. Lemma 1 highlights that, for the domain of the regular languages, it is better to generate accessible UNFAs than inaccessible UNFAs because the number of regular languages is not altered and the total number of NFAs is reduced. Table 5.9 shows that the three regular languages associated with the highest number of UNFAs make up 80 percent of the 622592 accessible UNFAs. These three regular languages, a^*, aa^* and the empty set, are also associated with 80 percent of the total number of UNFAs. However, the exclusion of inaccessible UNFAs results in a smaller number of UNFAs associated with each of the regular languages listed Table 5.9. This is the advantage of generating only accessible UNFAs.

Despite the exclusion of inaccessible NFAs, the number of NFAs associated with different regular languages varies significantly. Therefore, a new method is necessary for the generation of UNFAs across the domain of the regular languages. In the absence of such a method, we test to see if we can obtain improved results with the existing methods by studying density and final state selection, both of which are variables in the random generation methods. As the start state set is also a variable in the random generation methods, we initially briefly note the effect of the size of the start state set on the regular languages associated with the n-state UNFAs.

5.4.2 Start state set selection

The more start states there are in an UNFA, the more likely it is that the UNFA will be accessible. Each state must be reachable from a start state for the UNFA to be accessible. Definition 2, page 6, defines $q_0 \subseteq Q$ as the set of start states. Many methods to randomly generate UNFAs use a single start state [6, 18]. However, Van Zijl [29] generates a set of start states using a group of n bits. The probability of any specific state being a start state is then 50 percent. If we were to enumerate every possible NFA, all possible start state sets would be required. Therefore, when considering the structural domain, random start state sets would be required for randomly generating UNFAs with a uniform distribution. However, we want to test the effect of multiple start states on the domain of the regular languages.

Table 5.10 shows that, for three state UNFAs, the maximum possible number of regular languages can only be obtained with one start state. As soon as a start state set containing more than one element occurs, the number of regular languages which could be associated with the UNFA decreases. Therefore, this table leads us to expect that generating n-state UNFAs with varied number of elements in the start state set will have bad results over the domain of the regular languages. Experiment 20, page 77, confirms this. In the rest of our investigation of randomly

Number of	$\lvert Q_0\rvert = 1$	$\lvert Q_0\rvert = 2$	$\lvert Q_0\rvert = 3$
different regular languages	29	17	5
different RLs associated with minimal UNFAs	20	8	1

Table 5.10: The number of regular languages associated three state UNFAs with s start states.

generated UNFAs, we use only one start state. The example above shows that increasing the number of start states decreases the number of regular languages which can be associated with an n-state UNFA. Domaratzki *et. al.* [9] also restricted UNFAs to a single start state in their work.

5.4.3 Final state selection

In this section, we discuss the final state selection to use with the random NFA generation algorithms to attempt to improve the statistical performance of these methods across the domain of the regular languages. The results of these experiments are discussed in Section 5.4.5.

There are two ways we can measure the distribution of the final states across the domain of the regular languages for UNFAs.

1. The total number of final states may vary from zero to n. We can count the number of different regular languages associated with UNFAs with s final states, with $0 \leq s \leq n$.

2. We can also count the number of different regular languages which have the start state as a final state and compare this to the number of different regular languages which have state q_t as final, where q_t is not the start state.

The first point above should help us determine how to select the number of final states. The second point should help us determine which states should be chosen as final. We must note that in this case, although the start state is distinctive, all the other states can not be distinguished as there is no unique numbering of a UNFA. This means that for any regular language associated with a UNFA which has state q_t in the final state set, there is a numbering of states such that t can have any value other than the start state. Therefore, we can count the number of regular languages accepted by UNFA with the start state in the final state set and the number of regular languages accepted by UNFAs with at least one state other than the start state in the final state set. Due to the possible renumbering of states, the start state is the only recognisable state.

The number of different regular languages associated with n-state UNFAs with s final states

We can determine how the number of final states is to be chosen by analysing the number of different regular languages associated with n-state UNFAs with s final states. We have no enumeration of UNFAs or the regular languages accepted by UNFAs to derive an enumeration of the number of regular languages associated with UNFAs with s final states. For this reason we obtained Table 5.11 by experiment. All accessible n-state UNFAs were generated for both $n = 3$ and $n = 4$. Then the number of different languages for UNFAs with s final states was counted by a `bash` script. To obtain the number of different regular languages associated with minimal UNFAs with s final states, the regular languages not associated with $(n-1)$-state UNFAs were removed from the list of regular languages associated with n-state UNFAs with s final states. We note that there is no unique minimal UNFA. When referring to the different regular languages associated with minimal UNFAs, we are not concerned with the number of pairwise nonisomorphic minimal UNFAs. The focus of possible experiments is the number of different regular languages associated with minimal UNFAs.

Number of	$\|F\| = 0$	$\|F\| = 1$	$\|F\| = 2$	$\|F\| = 3$
different regular languages	1	20	15	3
RLs associated with minimal UNFAs	0	13	8	1

Number of	$\|F\| = 0$	$\|F\| = 1$	$\|F\| = 2$	$\|F\| = 3$	$\|F\| = 4$
different regular languages	1	55	57	26	4
RLs associated with minimal UNFAs	0	34	30	11	1

Table 5.11: The number of regular languages associated with three- and four state accessible UNFAs with s final states.

Table 5.11 can be used to simulate final state selection according to the distribution of final states for the domain of the regular languages. This may have little impact on the statistical performance as final states are not the only factor influencing the regular language associated with a UNFA.

The number of different regular languages which have state q_t as a final state, where t is less than n.

We can simulate the frequency of the number of final states for UNFAs. However, when we know the number of final states, the components of the final state set must still be chosen. In the UNFA case, we can count the number of regular languages associated with q_0 as a final state, where q_0 is the start state. All other states can not be distinguished, as there is no unique numbering of UNFAs. Therefore, we define Q_{other} as $Q - \{q_0\}$.

n=2	
q_0	5
Q_{other}	5

n=3	
q_0	15
Q_{other}	23

n=4	
q_0	44
Q_{other}	80

Table 5.12: The number of regular languages associated with final state sets including q_0 and Q_{other} for $n = 2, 3$ and 4.

Table 5.12 lists the number of regular languages associated with q_0 and Q_{other}. For $n = 2$ there is an equal probability of q_0 and q_t being final. As there are only two states, final states may be chosen uniformly. For $n = 3$, Q_{other} is part of the final state set of UNFAs associated with 23 regular languages and q_0 is part of the final state set for UNFAs associated with 15 regular languages. However, $Q_{other} = \{q_1, q_2\}$. As the number of states in the final state set vary, we

cannot say that each element in Q_{other} is in the final state set for UNFAs associated with $\frac{23}{2}$ regular languages. We do not have evidence from this data to suppose that the final states should not be uniformly chosen. We cannot say that as there are three elements in Q_{other} for $n = 4$, they are each associated with approximately $\frac{80}{3}$ regular languages, as they can all be final in the same UNFA. Furthermore, there is no unique minimal UNFA, so looking at minimal UNFAs is not helpful. If we look at the data in Table 5.12, we could conclude that q_0 should be final in half the generated UNFAs. However, there a 269 different regular languages associated with five state UNFAs, and only 130 of these have q_0 as part of the final state set. This means that we do not have enough evidence to support any specific method of choosing which states should form the final state set. Therefore, we select final states uniformly for UNFAs.

5.4.4 Transition density

Density is a required variable for both of Leslie's methods. The connected method generates UNFAs with a specific density. The all-density method produces UNFAs which will on average have the specified density, but individual UNFAs can have any transition density, depending on the random number stream. To have an idea of which density is optimal for the domain of the regular languages, we counted the number of different regular languages associated with UNFAs with each possible number of transitions for $n = 3$ and $n = 4$. (The density is a ratio of the number of transitions in an UNFA and the maximum number of possible transitions in the UNFA.) The maximum possible number of transitions in a UNFA is n^2, as there can be a transition from all of the n states to each of the states. Table 5.13 shows the number of different regular languages associated with UNFAs with zero to nine transitions for the 4096 three state UNFAs, (including pairwise isomorphic UNFAs and unconnected). Table 5.14, page 75, shows the distribution of regular languages associated with UNFAs which have transitions zero to 16 for the 1048576 four state UNFAs (including pairwise isomorphic UNFAs and unconnected UNFAs).

No. transitions	Density d(%)	No. of three state UNFAs with density d	No. regular languages associated with N	No. regular languages associated with minimal n-state UNFAs with density d
0	0	8	2	0
1	11	72	5	0
2	22	288	12	4
3	33	672	21	12
4	44	1008	17	8
5	55	1008	13	6
6	66	672	8	3
7	77	288	5	1
8	88	72	5	1
9	100	8	3	0

Table 5.13: Transition density in three state UNFAs and the number of regular languages associated with UNFAs with these densities.

As Experiment 15 on page 75 shows, UNFAs with transition densities of 31 percent, for $n = 4$, and 33 percent, for $n = 3$, have the most regular languages and the most distinct regular languages accepted by minimal UNFAs associated with them. (This is not the case for $\oplus$-UNFAs, see Chapter 6, page 89.) Transition density may be chosen according to these tables. However, the all-density method does not enforce a specific density. As previously mentioned, the average transition density in the generated UNFAs will be the specified density but some UNFAs will have different density. This must be taken into account when choosing a transition density for this method. For the all-density method, a constant density of 30 percent throughout the random generation has

No. transitions	Density d(%)	No. of four state UNFAs with density d	No. regular languages associated with N	No. regular languages associated with minimal n-state UNFAs with density d
0	0	16	2	0
1	6	256	5	0
2	13	1920	12	0
3	19	8960	30	8
4	25	29120	57	29
5	31	69888	63	34
6	38	128128	53	24
7	44	183040	40	13
8	50	205920	24	6
9	56	183040	19	3
10	63	128128	15	2
11	69	69888	10	1
12	75	29120	9	1
13	81	8960	6	1
14	88	1920	5	0
15	94	256	5	0
16	100.00	16	3	0

Table 5.14: Transition density in four state UNFAs and the number of regular languages associated with UNFAs with these densities.

reasonable results, as not all NFAs will have density of 30 percent. For an experiment in which the density is selected for every randomly generated NFA, we ignore densities below 30 percent in the tables because they will probably occur anyway with the all-density method. The connected method forces the required density so we can choose a varying transition density according to the transition density tables. We discuss the results of these experiments in Section 5.4.5.

5.4.5 UNFA experimental results

Experiment 14 *The bitstream method compared to Leslie's all-density method, with density chosen as* 50 *percent.*

As previously mentioned, the bitstream method (Algorithm 2, page 20), is equivalent to Leslie's all-density algorithm (Algorithm 4, page 26) with a 50 percent density. If this assertion is accurate, then the all-density method with 50 percent density and the bitstream method should produce similar results. Figure 5.14, page 82, shows the dotplots for $n = 4$ of 65000 UNFAs generated by the bitstream method and by the all-density method. As we expected, the dotplots are similar. For this reason, we will not use the bitstream method in any further UNFA experiments.

Experiment 15 *Leslie's all-density method, with density chosen as* 30 *percent,* 50 *percent and* 70 *percent.*

In Section 5.4.4, we discussed the transition density of the UNFA. The conclusion reached in that section was that the optimal density appears to be around 32 percent. Figure 5.15, page 82, shows 65000 four state UNFAs generated by the all-density method with densities of 30 percent, 50 percent and 70 percent. Although none of these dotplots are acceptable, the dotplot obtained by using the all-density method with a density of 30 percent is the best of the three dotplots. The dotplot obtained from the all-density method with a density of 50 percent was slightly worse than the dotplot obtained from the all-density method with a density of 30 percent as can be seen from the top right corner of the two dotplots. A density of 70 percent did not produce any integers

No. transitions	Density d (%)	No. of four state UNFAs with density d	No. regular languages associated with N	No. regular languages associated with minimal n-state UNFAs with density, d
0	0	0	0	0
1	6	0	0	0
2	13	0	0	0
3	19	256	16	8
4	25	2800	54	29
5	31	14016	63	34
6	38	42496	53	24
7	44	87040	40	13
8	50	127152	24	6
9	56	136320	19	3
10	63	108672	15	2
11	69	64512	10	1
12	75	28240	9	1
13	81	8896	6	1
14	88	1920	5	0
15	94	256	5	0
16	100	16	3	0

Table 5.15: Transition density in four state accessible UNFAs and the number of regular languages associated with UNFAs with these densities.

over 70. Therefore, this dotplot ranges over values zero to 70 instead the zero to 88. This shows a poor statistical performance across the domain of the regular languages.

Experiment 16 *Leslie's connected UNFA generation method for four state UNFAs with five transitions and eight transitions.*

Depending on the transition density (Table 5.14, page 75), we would expect the best results with five transitions for $n = 4$. Figure 5.16, page 83, shows the dotplot of 65000 UNFAs generated according to Leslie's connected method. The reason that this is better than the all-density method generating with a density of 31 percent, is that the number of transitions in each UNFA is exactly five. There are 63 regular languages associated with the 69888 possible different accessible UNFAs with five transitions. This results in a moderate distribution across the domain of the regular languages. The gaps in the graph are a result of the 25 regular languages with are not associated with UNFAs with five transitions. There are only 24 regular languages associated with a 50 percent density. This accounts for the poor dotplot of UNFAs with eight transitions. The first dotplot in Figure 5.16 is a better dotplot than Figure 5.15. This is because the all-density method does not produce a fixed density.

Experiment 17 *Leslie's all-density method, with density chosen as* 30 *percent and inaccessible UNFAs removed.*

The dotplot in Figure 5.17, page 83, illustrates 65000 four state UNFAs generated according to the all-density method, with a density of 30 percent. All inaccessible UNFAs where removed during the generation phase. This shows a slight improvement from the first plot in Figure 5.15. This can be attributed to the removal of the inaccessible UNFAs. Lemma 1, page 17, proves that all regular languages recognised by inaccessible n-state UNFAs are also recognised by accessible n-state UNFAs. This means that the removal of inaccessible UNFAs will not have an impact on the number of different regular languages it is possible to generate. In this experiment, there were 16465 UNFAs of the 65000 with a transition density higher than 50 percent. According to Table 5.14, there are only a small number of different regular languages accepted by minimal UNFAs

with transition densities greater than 50 percent.

Experiment 18 *Leslie's connected method with transition density selected according to the distribution of the different regular languages accepted by minimal UNFAs in Table 5.14.*

The left dotplot in Figure 5.18, page 83, shows the distribution of 65000 UNFAs over the domain of the regular languages. These UNFAs where generated with varying transitions, based on the distribution of the regular languages accepted by minimal UNFAs as given in Table 5.14. Final states for the dotplot on the left were chosen according to the bitstream method. Although this dotplot is not as good a distribution as Figure 5.16, there were 85 different regular languages recognised by the 65000 UNFAs. The maximum number of regular languages recognised by the accessible UNFAs generated with 30 percent density is 63. The second dotplot in Figure 5.18 shows the distribution of 65000 UNFAs generated with transition density selected according to Table 5.14 and final states selected according to the distribution of different regular languages accepted by minimal UNFAs with s final states. This is a slight improvement on the dotplot with bitstream final state selection.

Experiment 19 *Leslie's connected method with transition density selected according to the distribution of minimal UNFAs in Table 5.14 and the number of final states selected uniformly, with no empty or complete final state sets.*

The only difference between this experiment and the previous experiment is the final state selection. As can be seen from dotplot Figure 5.19, page 84, the results are similar to the results of the previous experiment. However, final state selection according to Table 5.11 seems to be slightly but not significantly better.

Experiment 20 *Multiple start states contrasted with a single, default start state.*

Figure 5.20, page 84, shows the dotplots of four state UNFAs with and without multiple start states across the domain of the regular languages. Leslie's all-density method was used for this experiment as the more start states, the higher the probability of an accessible UNFA. Transition density was selected according to the distribution of minimal UNFAs in Table 5.14, because this appeared to be one of the better methods of choosing transition density. Final states were selected according to the bitstream method. Start states were selected according to the bitstream method in the first plot and a single default start state in the second plot. Both are plots of 65000 UNFAs. The number of different regular languages obtained by the 65000 UNFAs with multiple start states was 68 as opposed to the 81 regular languages obtained from the UNFAs with a single, default start state.

5.4.6 Summary of results of UNFA experiments

Transition density chosen wisely can dramatically improve the distribution of the generated UNFAs over the domain of the regular languages. The optimal transition density appears to be around 30 percent. Final state selection does not appear to make that much difference, provided the empty final state set and complete final state set are limited.

5.5 Conclusion

We looked at the random generation of UDFAs across the domain of the regular languages. The method presented in Chapter 4 is functional for small and large values of n. In the case of binary DFAs, the bijection method appears to be the best approximation of the domain of the regular languages. However, for $n = 3$, there is still a clear clustering of points. The random generation of UNFAs across the domain of the regular languages cannot be solved by the separate study of transition density or final state selection. The optimal transition density for union-UNFAs is 30 percent.

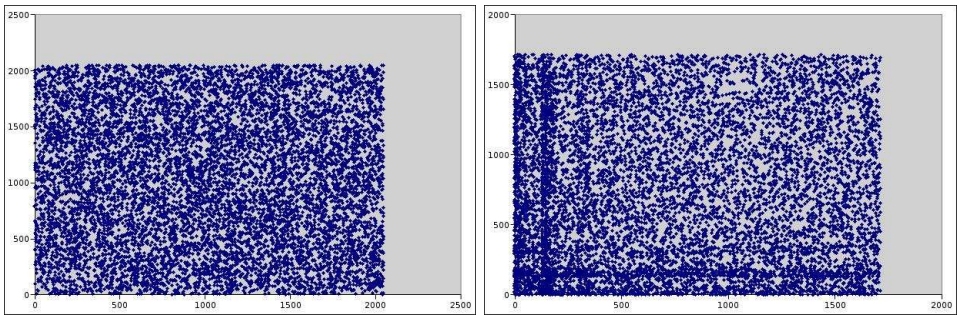


Figure 5.1: 10000 eight state UDFAs plotted across the domain of pairwise nonisomorphic accessible UDFAs (left) and the domain of the regular languages (right). These UDFAs were generated using Algorithm 1 and bitstream final states.

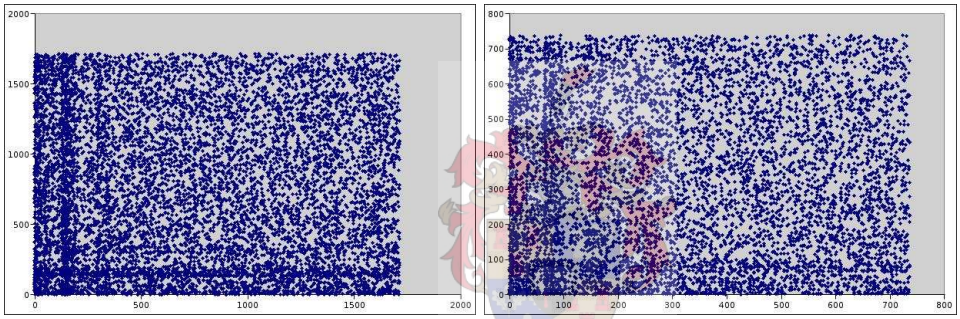


Figure 5.2: 10000 eight state UDFAs plotted across the domain of pairwise nonisomorphic accessible UDFAs (left) and the domain of the regular languages (right). These UDFAs were generated using Algorithm 1 and final states chosen as in Example 34.

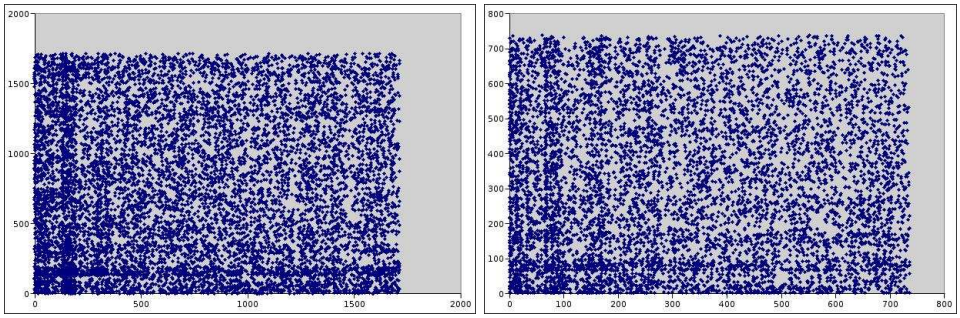


Figure 5.3: 10000 eight state UDFAs plotted across the domain of pairwise nonisomorphic accessible UDFAs (left) and the domain of the regular languages (right). These UDFAs were generated using Algorithm 1 and final states chosen as in Example 35.

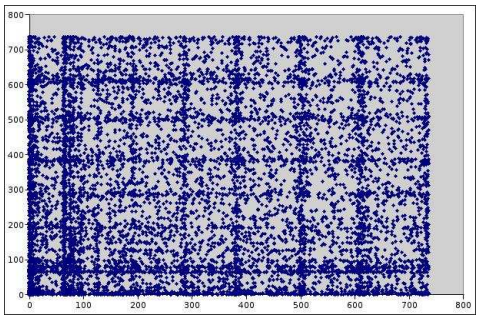
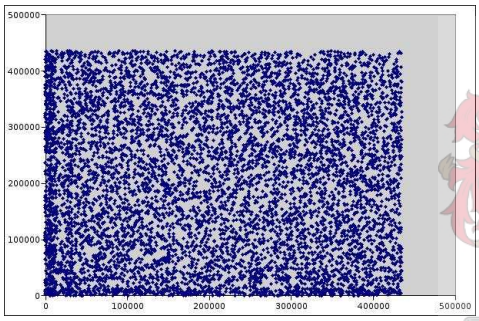


Figure 5.4: 8000 seven state UDFAs generated using Algorithm 1 and no empty final state sets or complete final state sets, plotted across the domain of the regular languages.



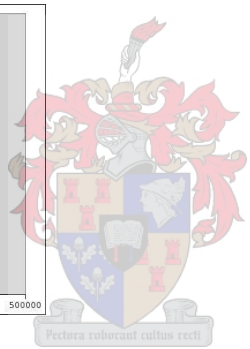
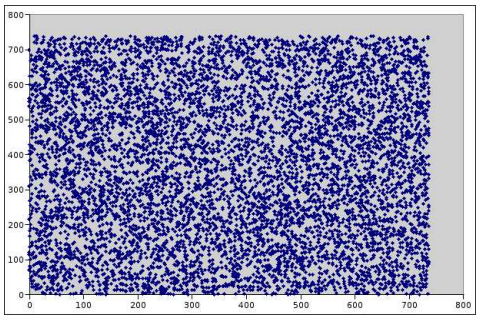


Figure 5.5: 20000 fifteen state UDFAs generated using Algorithm 1 and the bitstream method of final state selection, plotted across the domain of the regular languages.



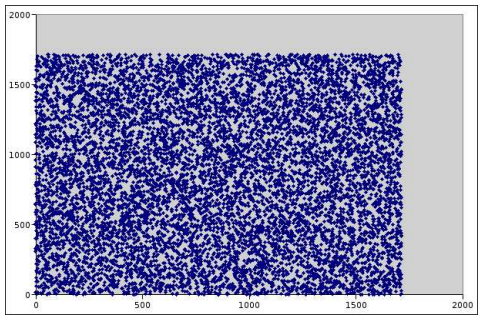


Figure 5.6: 8000 seven state UDFAs (left) and 10000 eight state UDFAs (right) generated according to the method described in Chapter 4, plotted across the domain of the regular languages.

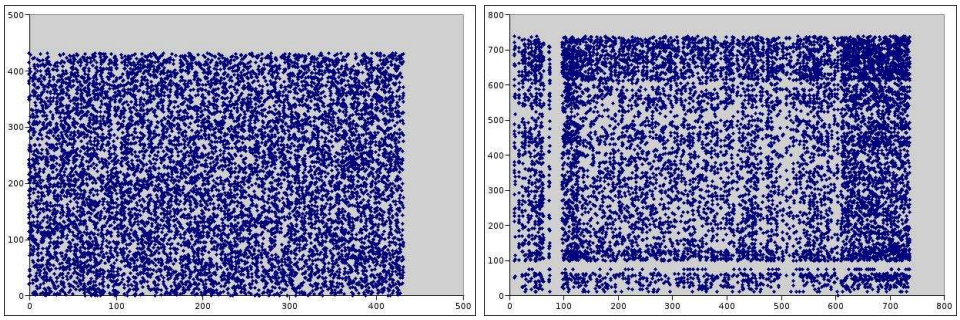


Figure 5.7: 10000 seven state minimal UDFAs plotted across the domain of the pairwise nonisomorphic minimal UDFAs (left) and plotted across the domain of the regular languages (right). The minimal UDFAs are generated according to the method to generate minimal UDFAs described in Chapter 4.

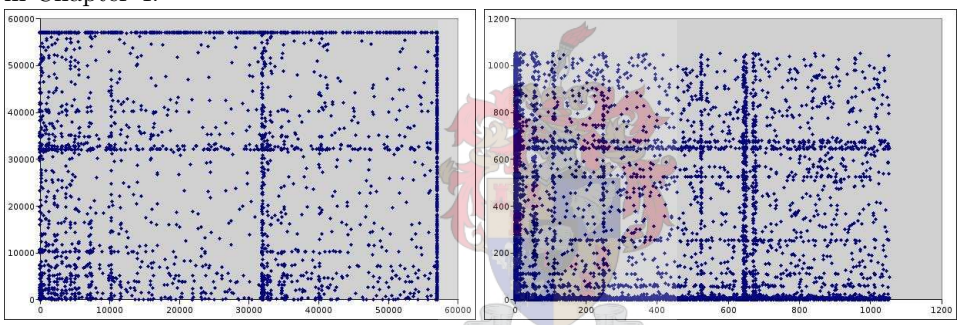


Figure 5.8: 3000 four state binary DFAs (left) and 10000 three state binary DFAs (right) plotted across the domain of the regular languages. The binary DFAs were generated using Algorithm 3 with final states chosen according to bitstream method.

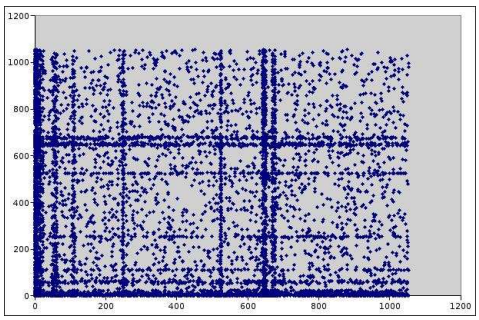


Figure 5.9: 10000 three state binary DFAs plotted across the domain of the regular languages. The binary DFAs were generated using Algorithm 3 with final states chosen uniformly, excluding the empty final set and $F = Q$.

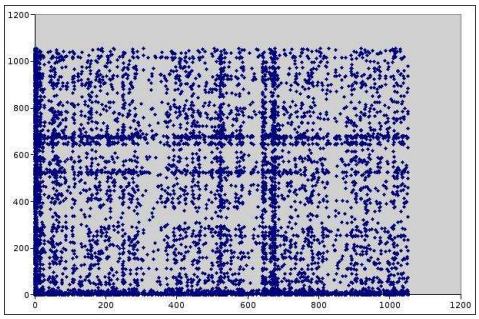


Figure 5.10: 10000 three state binary DFAs plotted across the domain of the regular languages. The binary DFAs were generated using Algorithm 6 with final states chosen according to bitstream method.

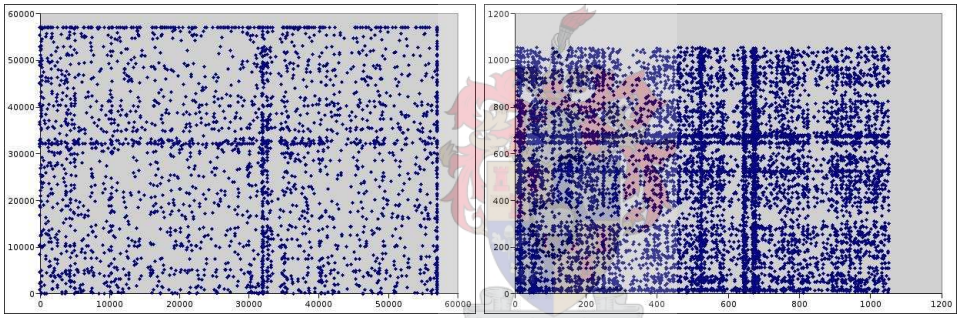


Figure 5.11: 3000 four state binary DFAs (left) and 10000 three state binary DFAs (right) plotted across the domain of the regular languages. The binary DFAs were generated using Algorithm 6 with final states chosen according to bitstream method of final state selection, but ignoring $F = \emptyset$ and $F = Q$.

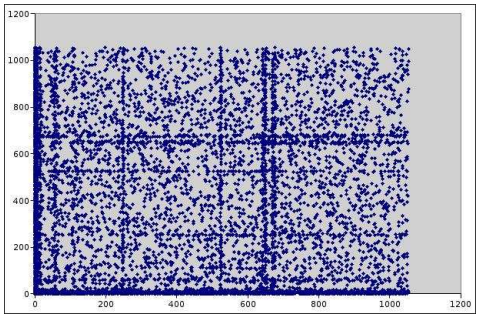


Figure 5.12: 10000 three state binary DFAs randomly generated according to the bijection method. Final states are selected according to the bitstream method.

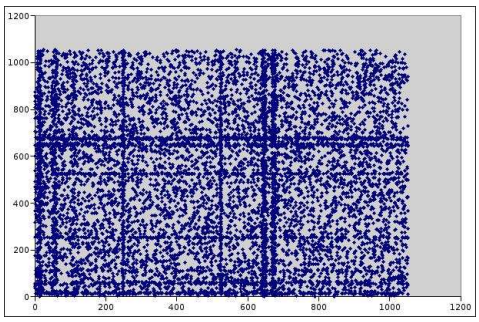


Figure 5.13: 10000 three state binary DFAs randomly generated according to the bijection method. Final states are selected uniformly, excluding the empty final state set and $F = Q$ as final state set.

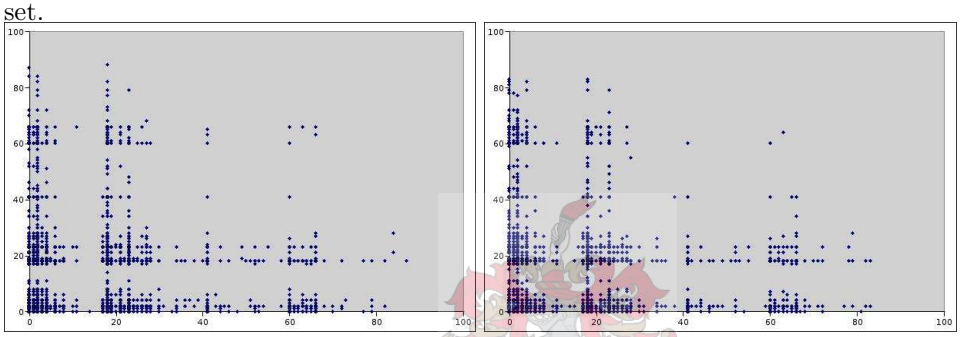


Figure 5.14: 65000 four state UNFAs randomly generated according to the bitstream method four state (left) and 65000 four state UNFAs randomly generated according to the all-density method with a 50 percent density (right) plotted across the domain of the regular languages.

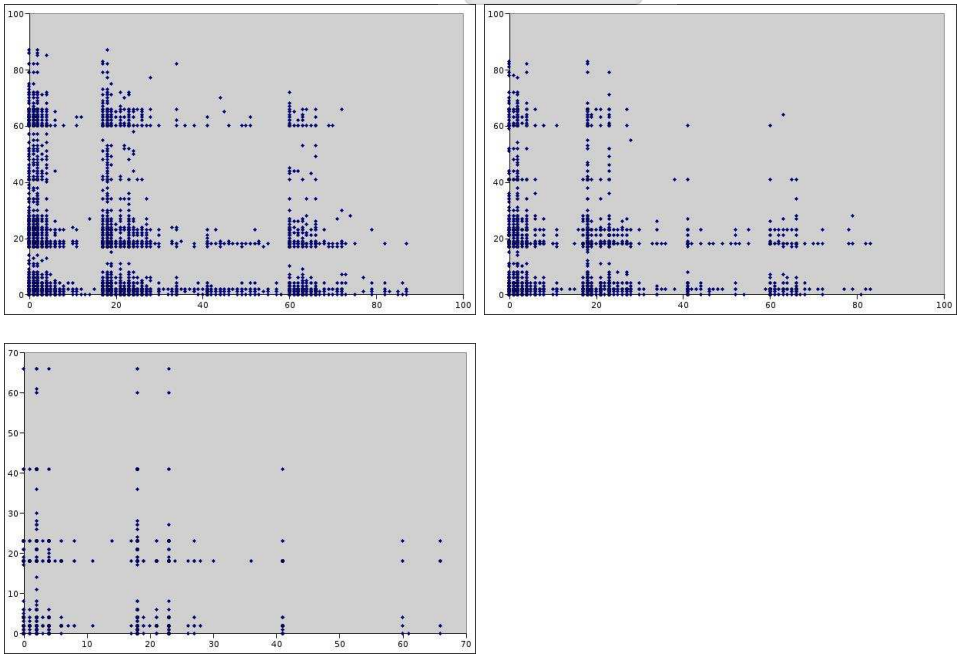


Figure 5.15: 65000 four state UNFAs randomly generated according to the all-density method with densities of 30 (left), 50 (right) and 70 (bottom), plotted across the domain of the regular languages.

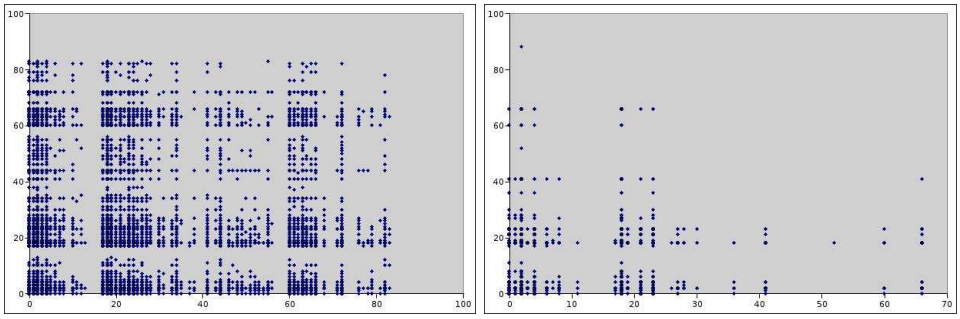


Figure 5.16: 65000 four state UNFAs randomly generated according to the connected method plotted across the domain of the regular languages. The UNFAs in the left plot each had five transitions (density of 31 percent) and the UNFAs in the right plot each had eight transitions (density of 50 percent).

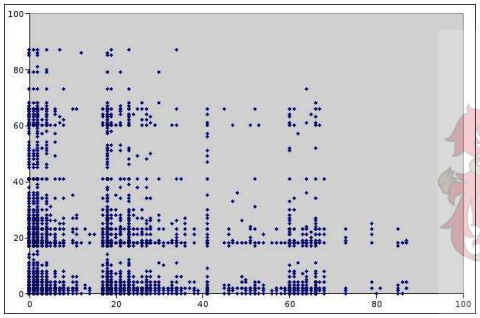


Figure 5.17: 65000 four state UNFAs randomly generated according to the all-density method with 30 percent density and inaccessible UNFAs removed, plotted across the domain of the regular languages.

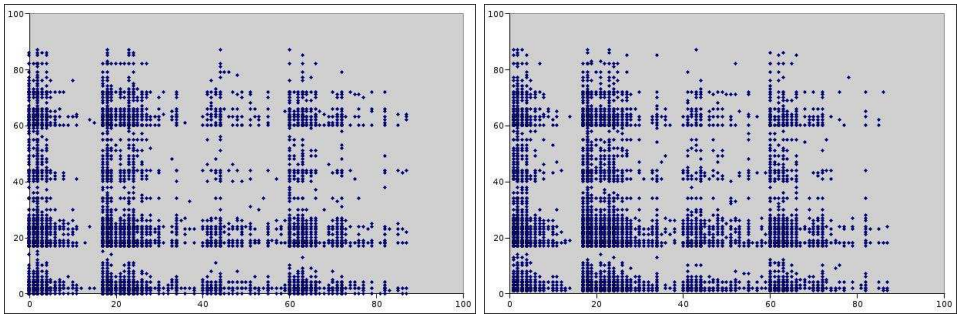


Figure 5.18: 65000 four state UNFAs randomly generated according to the connected method with transition density chosen according to the distribution of the number of regular languages accepted by minimal UNFAs in Table 5.14, page 75. The left plot has final states chosen according to the bitstream method and the right plot has final states chosen according to Table 5.11.

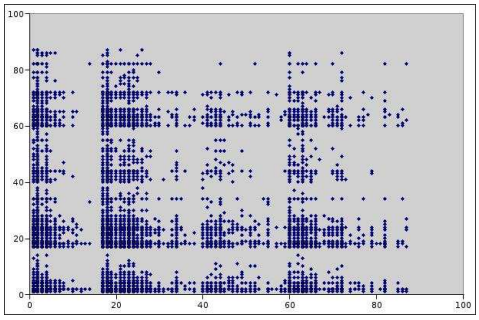


Figure 5.19: 65000 four state UNFAs randomly generated according to the connected method with transition density chosen according to the distribution of regular languages accepted by minimal UNFAs in Table 5.14, page 75. The number of final states is selected uniformly, excluding complete and empty final state sets.

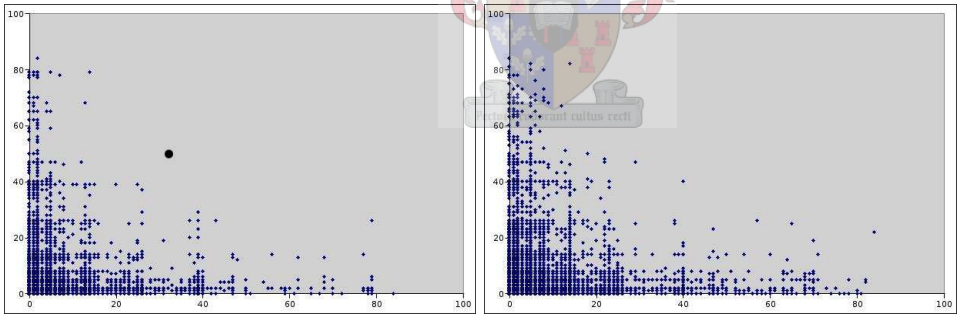


Figure 5.20: 65000 four state UNFAs randomly generated according to the all-density method, with final states selected according to the bitstream method, transition density chosen according to the distribution of the regular languages associated with minimal UNFAs in Table 5.14, page 75 and start states chosen according to the bitstream method (left). The second plot has final states and transition density chosen in the same manner, but a single start state.

Chapter 6

Random generation of $\oplus$-UNFAs

In Chapter 5, Section 5.4, we discussed the performance of the bitstream method and Leslie's *all-density* and connected methods over the domain of the regular languages for union-NFAs. In this chapter, we discuss the performance of these algorithms over the domain of the regular languages when they are used to generate $\oplus$-UNFAs. We also discuss transition density, start and final state selection and the regular languages associated with $\oplus$-UNFAs. Note that all $\oplus$-NFAs generated are restricted to a single start state except in Section 6.3, where we study the effect of different numbers of start states on the domain of the regular languages.

In Section 6.1, we discuss the experimental methods required to evaluate $\oplus$-UNFAs over the domain of the regular languages. We discuss the number of $\oplus$-UNFAs associated with various languages in Section 6.2. This gives us an indication of how the algorithms will perform over the domain of the regular languages. We discuss start state selection in Section 6.3, final state selection in Section 6.4 and transition density in Section 6.5. We comment on the number of minimal DFAs with different numbers of states which are equivalent to the n-state $\oplus$-UNFAs in Section 6.6. The experimental results are discussed in Section 6.7.

6.1 Experimental Methods

The experimental methods detailed in Chapter 5, Section 5.1.4, page 59, cannot be directly applied to $\oplus$-UNFAs as there is no available list of regular languages associated with n-state $\oplus$-UNFAs. However, the basic methods discussed in Section 5.1, page 56, still apply. We recapitulate them briefly below.

The randomly generated $\oplus$-UNFAs must be processed to obtain a stream of integers which represent the regular languages associated with the $\oplus$-UNFAs. Therefore, the elements in the domain of regular languages associated with n-state $\oplus$-UNFAs must be listed or enumerated and numbered. As no enumeration exists, a list of all the elements in the domain must be compiled and numbered. The number of the finite automaton in the list serves as the integer for that $\oplus$-UNFA wherever it occurs in the set of randomly generated finite automata. As previously mentioned, the distribution of the stream of integers is evaluated using a two-dimensional dotplot. We use the dotplots to analyse the distribution of randomly generated finite automata over the domain of the regular languages and over the domain of pairwise nonisomorphic finite automata.

Two steps are required to produce the stream of integers required for the dotplot. These steps are:

1. a list or an enumeration must be obtained for the domains and
2. the randomly generated finite automata must then be compared to the elements in the list.

6.1.1 Testing $\oplus$-UNFAs over the domain of the regular languages

There is no enumeration of the domain of the regular languages accepted by n-state $\oplus$-UNFAs. A list of the different regular languages can be obtained experimentally. We generate all possible accessible three and four state $\oplus$-UNFAs without the empty final state set or the final state set where $F = Q$. These $\oplus$-UNFAs are then converted to UDFAs using a standard routine to convert $\oplus$-UNFAs to UDFAs. The resulting UDFAs are minimised using Grail's `fmmin` routine. As Grail's `fmtore` routine produces a unique minimal regular expression for minimal UDFAs, we collect the list of regular expression associated with these UDFAs. This list of regular expressions can be compared using standard string operations to obtain the complete list of regular expressions. The empty language and $a^* + ""$ are added to the list of different regular expressions. These regular expressions represent the different regular languages accepted by n-state $\oplus$-UNFAs.

The randomly generated $\oplus$-UNFAs must be converted to UDFAs and minimised so that they can be directly compared to the minimised UDFAs in the lists. We can then use the `fmtore` routine to obtain regular expressions for all the UDFAs. These regular expressions can be compared to the list of different regular expressions representing the regular languages using standard string operations to generate the number stream.

6.2 Languages associated with n-state $\oplus$-UNFAs

From Equation 3.2, we know the total number of different possible $\oplus$-UNFAs to be $2^{n+m\times n^2}$ with m the number of alphabet symbols. Table 3.2, page 18, gives the number of different $\oplus$-UNFAs and the number of different languages associated with $\oplus$- and union-UNFAs for $n = 3$ and $n = 4$. This shows that there are many more regular languages associated with $\oplus$-UNFAs than regular languages associated with union-UNFAs. There are significantly more regular languages associated with n-state $\oplus$-NFAs than union-NFAs. This implies that the existing NFA generation algorithms are likely to produce better results over the domain of the regular languages for $\oplus$-NFAs than for the union-NFAs discussed in Section 5.4, page 70.

For $n = 4$, there are $2^n \times m \times 2^{n^2} = 1048576$ $\oplus$-UNFAs – including $\oplus$-UNFAs which are isomorphic to each other and $\oplus$-UNFAs which are not accessible. The bitstream method generates any of these $\oplus$-UNFAs with equal probability. The number of $\oplus$-UNFAs associated with each language shows how effective the random generations are likely to be over the domain of the regular languages. Table 6.1, page 87, lists the number of these 1048576 $\oplus$-UNFAs associated with specific regular expressions. The $\oplus$-UNFAs associated with a^*, the empty set and $a(aa)^*$ alone account for 50 percent of the 1048576 $\oplus$-UNFAs. Although this is still a high percentage, it is considerably better than the percentage of union-UNFAs associated with the three regular languages most commonly associated with union-UNFAs (80%). Table 5.9, page 71, lists the regular expressions for 20 different languages. There are 307 different languages associated with 4-state $\oplus$-UNFAs. The 20 in the table make up 78.5 percent of the $\oplus$-UNFAs. This means that there is a 21.5 percent probability that any of the other 296 languages will be associated with the generated $\oplus$-UNFAs. That is for every 1000 $\oplus$-UNFAs generated, we would expect 215 of them to be associated with the 296 languages not represented by the regular expressions in Table 5.9.

The number of possible $\oplus$-UNFAs which are accessible but include DFAs which are isomorphic to each other is 622592. Table 6.1, page 87, also shows the improvement over the number of $\oplus$-UNFAs associated with regular languages for inaccessible $\oplus$-UNFAs. However, the first three regular languages represented in the table, $a^* + ""$, the empty set and aa^* still comprise forty percent of the 622592 accessible $\oplus$-UNFAs. Assuming we remove the empty final state set which is only associated with the empty set and the final state set where $F = |Q|$ which is only associated with $a^* + ""$, which each make up 38912 of the total 622592 $\oplus$-UNFAs, then the first three regular languages in the table would only make up 27 percent of the accessible $\oplus$-UNFAs. However, this is

regular expression	no. of total $\oplus$-UNFAs	no. accessible $\oplus$-UNFAs
$a^* + ""$	257680	152720
empty set	156672	41984
aa^*	69824	49088
""	47680	4928
$a(aaa)^* + a(aaa)^*a$	37760	21632
$a + ""$	30432	13152
$(aa)^*a$	29888	16064
a	28160	14336
$a(a^7)^* + a(a^7)^*a + a(a^7)^*aa + a(a^7)^*aaa$		
$+a(a^7)^*aaaaa + a(a^7)^*aaaa$	19584	14976
$a + aa + ""$	17232	11280
$aa + a$	13056	9216
$(aaa)^* + (aaa)^*a + ""$	12928	6400
$aa(aaa)^* + aa(aaa)^*a + ""$	12208	5680
$a(aaaa)^* + a(aaaa)^*aa + a(aaaa)^*a$	12096	9792
$a(aa)^*a + ""$	9760	4000
$a + aa + aaa + ""$	6552	6552
aaa^*	6336	4800

Table 6.1: The number of structurally different unconnected 4-state $\oplus$-UNFAs (including all isomorphic $\oplus$-UNFAs) associated with selected regular expressions.

still a problem because there are 51 other regular languages associated with four state $\oplus$-UNFAs. Although these values are better that the union-NFA case, it is still clear that a new method is necessary for the generation of $\oplus$-UNFAs across the domain of the regular languages. We test to see if we can obtain improved results with the existing methods by studying density and final state selection.

6.3 Start state selection

In the case of union-UNFAs, there was a significant decrease in the number of regular languages associated with UNFAs with more than one start state. The experiment is done by counting all regular languages associated with all possible UNFAs (including inaccessible UNFAs) with s start states. This is because each state must be reachable from a start state for the UNFA to be accessible. Thus UNFAs with n start states are always accessible.

As previously mentioned, many methods to randomly generate UNFAs use a single start state [6, 18]. If we were to enumerate every possible UNFA, all possible start state sets would be required. Therefore, when considering the structural domain, random start state sets would be required for randomly generating UNFAs with a uniform distribution. However, we want to test the effect of multiple start states on the domain of the regular languages.

Table 6.2, page 88, shows that although the maximum number of regular languages possible can only be obtained with one start state, the decrease is far less significant for increasing start states than in the union-UNFA case (Table 5.10, page 72). This is what we would expect based on the concept of symmetric difference. In the union case, according to the subset construction, once we have a state labelled according to the maximum number of states, assuming there is a transition to every one of those states from another state, the state label is not going to decrease. However, with symmetric difference, the number of states forming the label can decrease and increase again. Although the difference is far less significant, we still generated $\oplus$-UNFAs with one start state for the rest of our investigation of randomly generated $\oplus$-UNFAs.

Number of	$\|Q_0\| = 1$	$\|Q_0\| = 2$	$\|Q_0\| = 3$
different regular languages	54	52	33
different RLs associated with minimal $\oplus$-UNFAs	43	41	25

Table 6.2: The number of regular languages associated three state $\oplus$-UNFAs with s start states.

6.4 Final state selection for $\oplus$-UNFAs

In this section, we discuss the final state selection to use with random generation algorithms to attempt to improve the statistical performance of these methods across the domain of the regular languages accepted by n-state $\oplus$-UNFAs.

As previously mentioned, there are two ways we can measure the distribution of the final states across the domain of the regular languages for $\oplus$-UNFAs.

1. The total number of final states may vary from zero to n. We can count the number of different regular languages associated with $\oplus$-UNFAs with s final states, with $0 \leq s \leq n$.

2. We can also count the number of different regular languages which have the start state as a final state and compare this to the number of different regular languages which have state q_t as final, where q_t is not the start state.

6.4.1 The number of different regular languages associated with n-state $\oplus$-UNFAs with s final states

We can determine how the number of final states is to be chosen by analysing the number of different regular languages associated with n-state $\oplus$-UNFAs with s final states. We obtained the values for four state $\oplus$-UNFAs in Table 6.3 by experiment as we have no enumeration. The experiment was equivalent to the experiment to determine the number of regular languages associated with union-UNFAs which have a final state set of size s, therefore, we do not describe the methods here.

Number of	$\|F\| = 0$	$\|F\| = 1$	$\|F\| = 2$	$\|F\| = 3$	$\|F\| = 4$
different regular languages	1	170	194	65	4
different minimal $\oplus$-UNFAs with s final states	0	130	156	41	1

Table 6.3: The number of different languages associated with s final states and the number of different regular languages associated with $\oplus$-NFAs which have s final states, for $n = 4$.

Table 6.3 can be used to select the number of final states in the final state set. Once again, the final state selection can result in an improvement but the regular language associated with the $\oplus$-UNFA depends on the transition table, start states and final states, so final state selection independent to the transition table generation will never be a total solution to the random generation of $\oplus$-UNFAs over the domain of the regular languages.

6.4.2 The number of different regular languages which have state q_t as a final state, where t is less than n.

As previously mentioned, the start state is the only distinctive state in the $\oplus$-UNFAs as there is no unique number on $\oplus$-UNFAs. Therefore, we are only able to uniquely identify the start state. The start state is identifiable because we only have one start state. We can also test the number of other languages which have state q_t, where $t \neq q_0$, in the final state set. We go through all possible $\oplus$-UNFAs which implies that all possible numberings of the $\oplus$-UNFAs will be considered. Therefore, all states other than the start state will have an equal number of different regular

languages associated with them. Thus, we define Q_{other}, representing all states other than the start state, as $Q - \{q_0\}$.

n=2	
q_0	6
Q_{other}	6

n=3	
q_0	31
Q_{other}	41

n=4	
q_0	186
Q_{other}	274

Table 6.4: The number of regular languages associated with q_0 and Q_{other} for $n = 2, 3$ and 4.

Table 6.4 lists the number of regular languages associated with q_0 and Q_{other}. For $n = 2$ there is an equal probability of q_0 and q_t being final. As there are only two states, final states may be chosen uniformly. For $n = 3$ and $n = 4$, it is difficult to determine the role of Q_{other} as not only is there more than one state in the set Q_{other}, but more than one of these states can be final at the same time. However, it appears from the number of regular languages associated with four state $\oplus$-UNFAs with q_0 in the final state set that q_0 should be final $\frac{186}{307} \times 100 = 60$ percent of the time. Overall, this information is not helpful in choosing final states.

6.5 Transition density for $\oplus$-NFAs

As density is a required variable for both of Leslie's methods, we need to choose densities to obtain the best results over the domain of the regular languages. To have an indication of which density is optimal for the domain of the regular languages, we counted the number of different regular languages associated with $\oplus$-UNFAs with each possible number of transitions for $n = 3$ and $n = 4$. (The density is a ratio of the number of transitions in an $\oplus$-UNFA and the maximum number of possible transitions in the $\oplus$-UNFA.) A density of 100 percent, requires all possible n^2 transitions. Table 6.5shows the number of different regular languages associated with $\oplus$-UNFAs with zero to nine transitions for the 4096 three state $\oplus$-UNFAs, (including pairwise isomorphic $\oplus$-UNFAs and unconnected $\oplus$-UNFAs). Table 6.6, page 90, shows the distribution of regular languages associated with $\oplus$-UNFAs which have transitions zero to 16 for the 1048576 four state $\oplus$-UNFAs (including pairwise isomorphic $\oplus$-UNFAs and unconnected $\oplus$-UNFAs).

Number of transitions	0	1	2	3	4	5	6	7	8	9
Total number of unconnected $\oplus$-UNFAs	8	72	288	672	1008	1008	672	288	72	8
Density (%)	0	11	22	33	44	55	66	77	88	100
Languages associated	3	5	12	20	37	43	39	32	15	3
Minimal languages	0	0	4	9	27	33	29	23	7	0

Table 6.5: Transition density in three state $\oplus$-UNFAs and the number of regular languages associated with $\oplus$-UNFAs with these densities.

The $\oplus$-UNFAs with transition densities of approximately 50 percent are associated with the most regular languages and the most distinct regular languages accepted by minimal $\oplus$-UNFAs. This means that the bitstream method should be used or the equivalent all density method with a density of 50 percent. (Note that the optimal density for union-UNFAs is near 30 percent, see Chapter 5, page 74.) Transition density may be chosen according to these tables. We discuss the experimental results in Section 6.7. These results include experiments with various transition densities.

6.6 DFAs

It can be of interest to study the number of minimal DFAs with different numbers of states which are equivalent to the n-state $\oplus$-UNFAs. Domaratzki *et. al.* [9] provide this information for

No. transitions	Density d(%)	N, no. of 4-state $\oplus$-UNFAs with density, d	No. languages associated with N	No. languages associated with minimal $\oplus$-UNFAs with density d
0	0	16	2	0
1	6	256	5	0
2	13	1920	12	0
3	19	8960	30	8
4	25	29120	69	33
5	31	69888	170	132
6	38	128128	248	192
7	44	183040	301	244
8	50	205920	307	250
9	56	183040	307	250
10	63	128128	303	246
11	69	69888	292	237
12	75	29120	246	206
13	81	8960	103	67
14	88	1920	35	19
15	94	256	9	0
16	100	16	3	0

Table 6.6: Transition density in four state $\oplus$-UNFAs and the number of regular languages associated with $\oplus$-UNFAs with these densities.

union-UNFAs. Table 6.7 lists the number of complete DFAs with j states that are equivalent to $\oplus$-UNFAs with n-states. The lack of five and six state DFAs equivalent to three state $\oplus$-UNFAs is consistent with the results in [28]. If this table could be generated, it could lead to an enumeration of the regular languages associated with $\oplus$-UNFAs.

	j=1	j=2	j=3	j=4	j=5	j=6	j=7
n=1	2	1	–	–	–	–	–
n=2	2	4	5	–	–	–	–
n=3	2	4	12	12	–	–	21

Table 6.7: The number of complete minimal DFAs with j states equivalent to $\oplus$-UNFAs with n states.

6.7 $\oplus$-UNFA experimental results

Experiment 21 *$\oplus$-UNFAs randomly generated according to Leslie's all-density method, with density chosen as* 30 *percent,* 50 *percent and* 70 *percent.*

We discussed the transition density of the $\oplus$-UNFA over the domain of the regular languages in Section 6.5. Symmetric difference UNFAs have an optimal density of 50. Union UNFAs have an optimal density of 30 percent. Therefore, in this experiment we plotted the $\oplus$-UNFAs over the domain of the regular languages with densities of 30, 50 and 70 percent in Figure 6.1, page 93. As we would expect from Table 6.6, page 90, $\oplus$-UNFAs with a density of 30 percent result in a poor dotplot. As can be seen from the table, there are only 170 regular languages associated with $\oplus$-UNFAs which have a density of 30 percent. There is less difference between the plotted $\oplus$-UNFAs with a density of 50 and those with a density of 70 percent. There are 307 different regular languages associated with $\oplus$-UNFAs with a density of 50 percent, 292 regular languages

associated with a density of 69 percent and 246 regular languages associated with $\oplus$-UNFAs with a density of 75 percent. Although the difference is not great, the dotplot of $\oplus$-UNFAs randomly generated with a density of 70 percent is darker near the axes than the dotplot of $\oplus$-UNFAs randomly generated with a density of 50 percent.

Experiment 22 *$\oplus$-UNFAs randomly generated according to Leslie's all-density method, with density chosen as 50 percent and inaccessible $\oplus$-UNFAs removed.*

Figure 6.2, page 93, shows 65000 four state $\oplus$-UNFAs randomly generated according to the all-density method, with a density of 50 percent plotted across the domain of the regular languages. All inaccessible $\oplus$-UNFAs where removed during the random generation phase. There is a significant improvement, which can be seen by comparing this figure to the dotplot which has inaccessible $\oplus$-UNFAs included in the randomly generated set of $\oplus$-UNFAs (Figure 6.1, page 93). The dotplot shows an improved distribution, as the whole area of the dotplot is more densely covered than in Figure 6.1. This can be attributed to the removal of the inaccessible $\oplus$-UNFAs.

Experiment 23 *$\oplus$-UNFAs randomly generated according to Leslie's accessible method with eight and nine transitions.*

Figure 6.3, page 93 illustrates the similarity of the results for four state $\oplus$-UNFAs generated with eight and nine transitions respectively. Our implementation of this method forced the number of transitions to exactly the eight or nine transitions. This did not result in an overall improvement as Figure 6.2, the plot of $\oplus$-UNFAs generated according to the all-density method, with inaccessible UNFAs removed, has a better distribution than either of the plots in Figure 6.3.

Experiment 24 *$\oplus$-UNFAs randomly generated according to Leslie's accessible method, with transitions chosen according to the number of regular languages associated with each density, from Table 6.6.*

The previous experiment showed that a varied density as in the all-density method with inaccessible UNFAs removed has better results than the optimal density as a fixed density. Therefore, Figure 6.4, page 94, shows the dotplot over the domain of the regular languages of UNFAs with a varied density. This figure is at least as uniform as Figure 6.2, the all-density method with the inaccessible UNFAs removed, without the extra processing required to remove the inaccessible UNFAs.

Experiment 25 *$\oplus$-UNFAs randomly generated according to Leslie's accessible method, with transitions chosen according to the number of regular languages associated with minimal UNFAs for each density, from Table 6.6. To this is added, in a second plot, final states selected according to the regular languages associated with minimal UNFAs for each different size of final state set, Table 6.3, page 88.*

As can be seen from Figure 6.5, page 94, the selection of density according to the number of regular languages associated with minimal UNFAs for each density results a standard distribution in the dotplot. The inclusion of a special method of final state selection results in a significant improvement. The dotplot on the right has the best distribution we obtained experimentally for $\oplus$-UNFAs over the domain of the regular languages.

Experiment 26 *$\oplus$-UNFAs randomly generated according to Leslie's all-density method, with a transition density of 50 percent. The plot on the left have a single default start state, while the plot on the right has start states chosen according to the bitstream method.*

Figure 6.6, page 94, shows that multiple start states have a negative effect on the distribution of $\oplus$-UNFAs over the domain of the regular languages. This is expected, based on the data in Table 6.2, page 88.

6.8 Conclusion

The random generation of $\oplus$-UNFAs across the domain of the regular languages cannot be solved by the separate study of transition density or final state selection. However, the optimal transition density for $\oplus$-UNFAs is 50 percent. This corresponds to Champarnaud *et. al*'s results relating to $\oplus$-NFAs in [6]. Furthermore, modified final state selection to limit the occurrences of final state sets $F = Q$ and the empty final state set result in a noticeable improvement. There are significantly more regular languages associated with $\oplus$-UNFAs than with union-UNFAs.

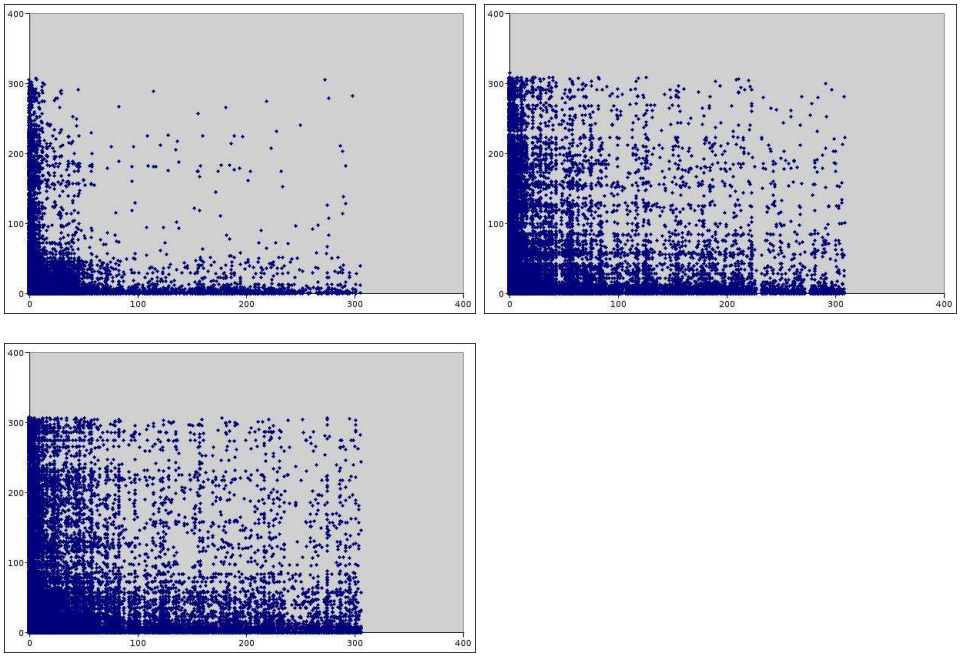


Figure 6.1: 65000 four state $\oplus$-UNFAs randomly generated according to the all-density method, with densities of 30 (left), 50 (right) and 70 (bottom), plotted across the domain of the regular languages.

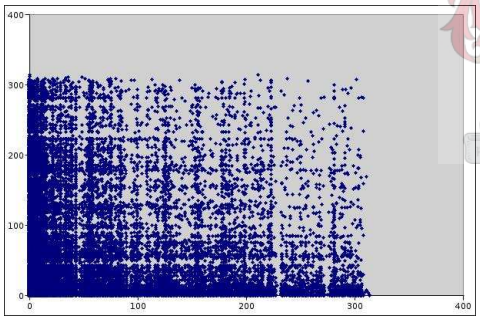


Figure 6.2: 65000 four state $\oplus$-UNFAs randomly generated according to the all-density method, plotted across the domain of the regular languages. Inaccessible $\oplus$-UNFAs have been removed.

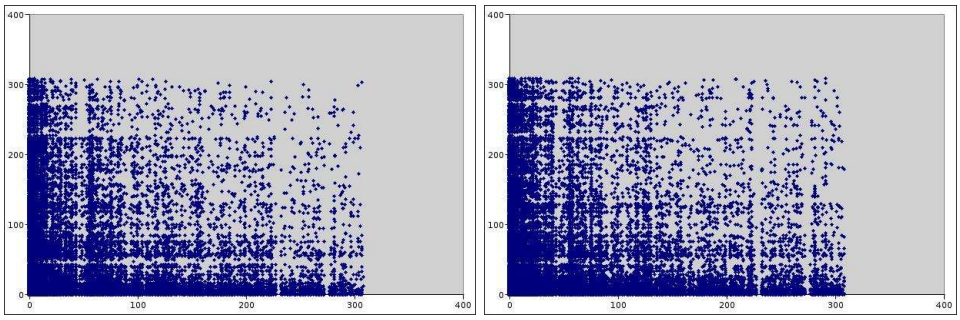


Figure 6.3: 65000 four state $\oplus$-UNFAs randomly generated according to the accessible method, plotted across the domain of the regular languages. In the plot on the left, each $\oplus$-UNFA had eight transitions and in the plot on the right, each $\oplus$-UNFA had nine transitions.

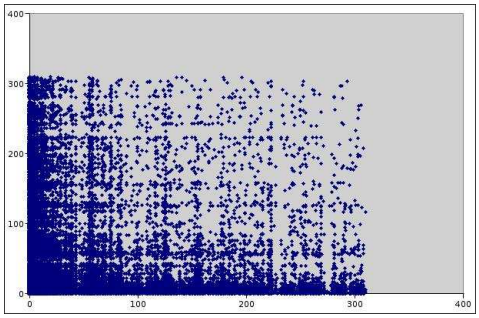


Figure 6.4: 65000 four state $\oplus$-UNFAs randomly generated according to the accessible method, with transition densities chosen according to the distribution of the regular languages in Table 6.6, plotted across the domain of the regular languages.

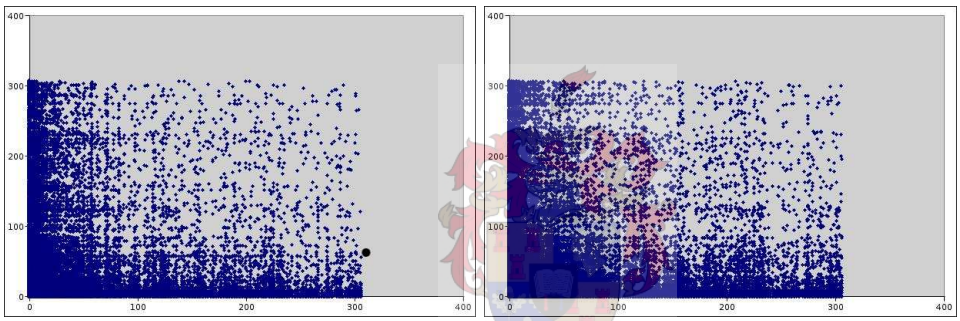


Figure 6.5: 65000 four-state $\oplus$-UNFAs generated according to the accessible method, plotted across the domain of the regular languages. Transitions selected according to the number of regular languages associated with minimal UDFAs of specific densities, Table 6.6. In the plot on the left, final states were selected according to the bitstream method. In the plot on the right, final states were chosen according to the distribution of the regular languages associated with minimal UNFAs, Table 6.3.

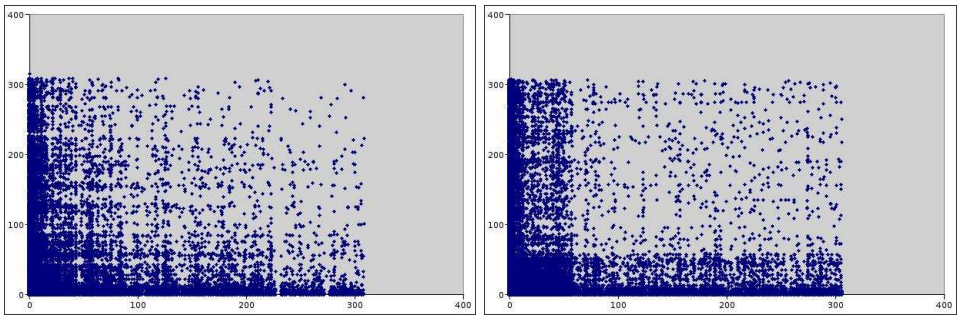


Figure 6.6: 65000 four state $\oplus$-UNFAs generated according to the all-density method with a density of fifty percent, plotted across the domain of the regular languages. The UNFAs in the left plot have a single, default start state. The plot on the right has start states chosen according to the bitstream method.

Chapter 7

Conclusion

In this thesis, we investigated the uniform random generation of finite automata over the domain of the regular languages. We looked initially at enumerations of finite automata with certain restrictions. Then we studied the performance of existing algorithms for the random generation of finite automata over the domain of the regular languages. For small values of n, all the algorithms evaluated in this study were unsatisfactory. The random generation of n-state finite automata for small values of n is important because patterns can be determined through a visual examination and the sample size is not overwhelming.

For UDFAs, we derived an algorithm for the random generation of UDFAs over the domain of the regular languages from Domaratzki *et. al.*'s enumeration of the domain of the regular languages. This algorithm randomly generates UDFAs uniformly over the domain of the regular languages for any value of n.

The conclusion to our study of the random generation of binary DFAs over the domain of the regular languages showed that the bijection method would be suitable where the value of n is large, if the empty final state set and the final state set such that $F = Q$ are restricted. The dotplot of three state binary DFAs over the domain of the regular languages exhibited some clusters of points indicating non-uniform behaviour. However, the dotplot, combined with the numeric evidence of the number of pairwise nonisomorphic DFAs, compared to the number of regular languages associated with them, led us to believe that the bijection method will have good results over the domain of the regular languages for large n.

The existing algorithms to randomly generate $\oplus$-UNFAs and $\cup$-UNFAs over the domain of the regular languages do not produce uniform results over the domain of the regular languages for any value of n. Manipulation of start and final state selection as well as transition density result in improvement in the distribution. However, that improvement is not sufficient to yield adequate results.

The results from the existing algorithms to randomly generate UNFAs proved without a doubt that an algorithm for the random generation for $\oplus$-NFAs and $\cup$-NFAs is required. An enumeration of the languages associated with $\oplus$-UNFAs and $\cup$-UNFAs would be a solid foundation for a random generation algorithm.

Bibliography

[1] T. M. Apostol. *Introduction to Analytic Number Theory.* Springer-Verlag, New York, 1976.

[2] F. Bassino and C. Nicaud. Enumeration of complete accessible deterministic automata over a 2-letter alphabet. Withdrawn, available at http://www.cs.sun.ac.za/∼lraitt/.

[3] F. Bassino and C. Nicaud. Enumeration and random generation of accessible automata. 2006. Preprint, available at http://igm.univ-mlv.fr/ bassino/publi.html.

[4] Noud De Beijer, Bruce W. Watson, and Derrick G. Kourie. Stretching and jamming of automata. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on enablement through technology*, pages 198–207, Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[5] B. W. Brown and J. Lovato. Ranlib. http://www.stat.umn.edu/HELP/ranlib-docs/ranlib.fdoc, 1991.

[6] J-M. Champarnaud, G. Hansel, T. Paranthoen, and D.Ziadi. Random generation models for NFAs. *Journal of Automata Languages and Combinatorics*, 2004.

[7] J-M. Champarnaud and T. Paranthoën. Random generation of DFAs. *Theoretical Computer Science*, 330(2):221–235, 2005.

[8] L.J. Cummings. Shuffled Lyndon Words. *ARS COMBINATORIA*, 33:47–56, 1992.

[9] M. Domaratzki, D. Kisman, and J. Shallit. On the number of distinct languages accepted by finite automata with n states. *Journal of Automata Languages and Combinatorics*, 7(4):469–486, 2002.

[10] D.Raymond and D.Wood. The user's guide to grail. http://www.csd.uwo.ca/research/grail, 1996.

[11] F. Harary and E. Palmer. Enumeration of finite automata. *Information and Control*, 10(5):499–508, 1967.

[12] M. A. Harrison. A census of finite automata. *Canadian journal of Mathematics*, 17:100–113, 1965.

[13] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1981. Vol. 2, 2nd ed.

[14] A. M. Law and W. D. Kelton. *Simulation Modeling & Analysis*, chapter 7, pages 420–461. McGraw-Hill, Inc., New York, 1991.

[15] P. L'Ecuyer. Testing random number generators. *Proceedings of the 1992 Winter Simulation Conference*, pages 305–313, 1992.

[16] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[17] P. L'Ecuyer. *Handbook on Simulation*, chapter 4. Wiley, New York, 1998.

[18] T. Leslie. *Efficient Approaches to Subset Construction.* Master's thesis. University of Waterloo, Ontario, Canada, 1995.

[19] M.Matsumoto and T.Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulations*, 8(1):3–30, 1998.

[20] C. Nicaud. Average state complexity of operations on unary automata. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, pages 231–240. Springer-Verlag, 1999.

[21] C. Nicaud. *Etude du comportement en moyenne des automates finis et des langages rationnels,*Ph.D. thesis. University of Paris, France, 2000.

[22] C.E. Radke. Enumeration of strongly connected sequential machines. *Information and Control*, 8:377–389, 1965.

[23] R. W. Robinson. Counting strongly connected finite automata. In *Graph theory with applications to algorithms and computer science*, pages 671–685, New York, USA, 1985. John Wiley & Sons, Inc.

[24] F. Ruskey. Necklaces, lyndon words, debruijn sequences. http://theory.cs.uvic.ca/cos.html, 2000.

[25] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[26] S.J. Steele. Private comunications. *University of Stellenbosch, Mathematical Statistics Deparartment*, 2005.

[27] A. P. Streeet and W.D. Wallis. *Combinatorial theory: an introduction.* The Charles Babbage Research Center, Winnipeg, Canada, 1997.

[28] L. van Zijl. *Generalized Nondeterminism and the Succinct Representation of Regular Languages.* Ph. D. thesis. University of Stellenbosch, Stellenbosch, South Africa, 1997.

[29] L. van Zijl. Random Number Generation with Symmetric Difference NFAs. *Proceedings of 6th International Conference on the Implementation and Application of Automata (CIAA2001)*, July 2001.

[30] L. van Zijl, F. Olivier, and J.-P. Harper. The MERLin Environment applied to *-NFAs. *Proceedings of the CIAA*, July 2000.